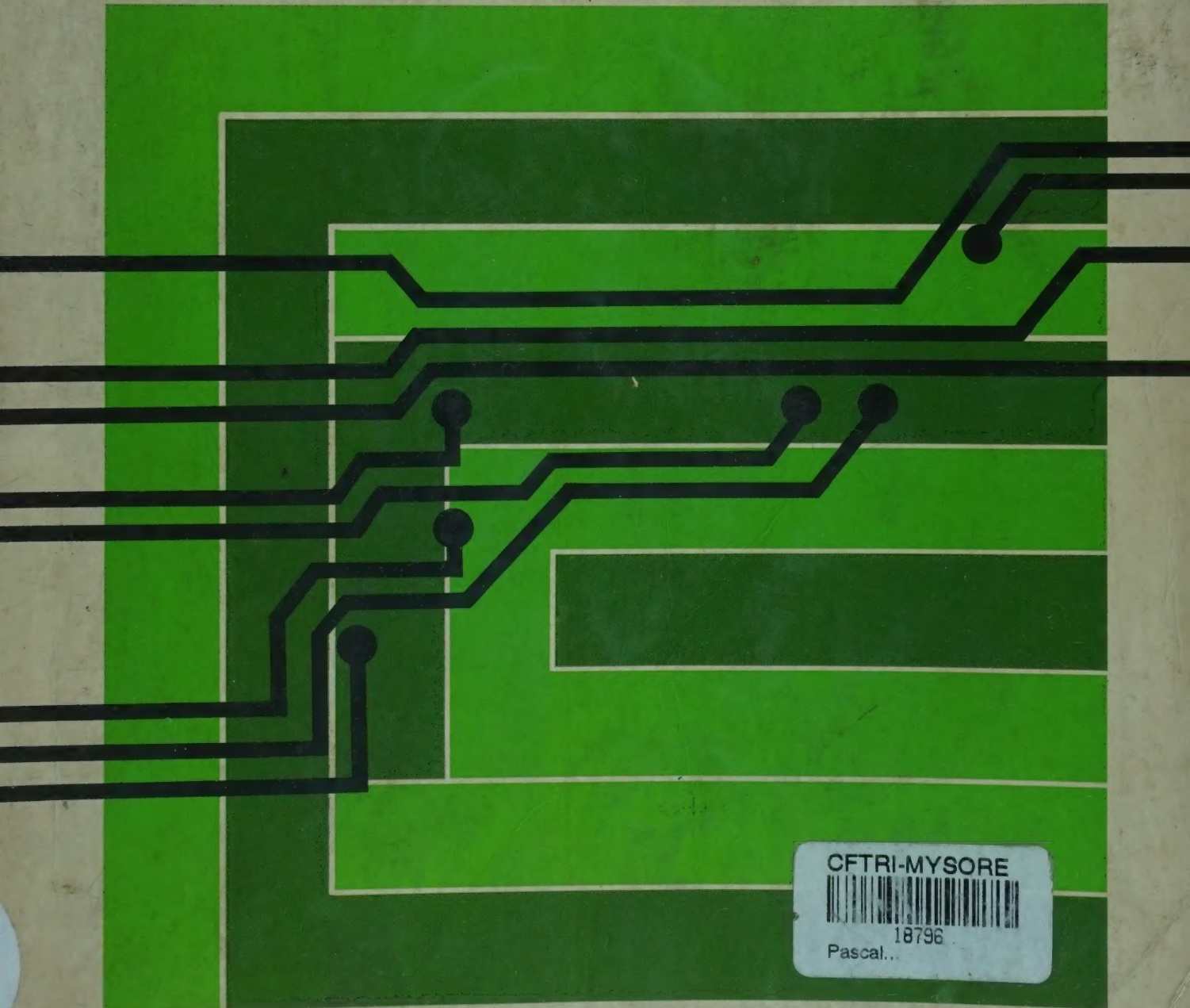
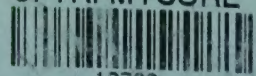


PASCAL

GENEVA G. BELFORD AND C. L. LIU



CFTRI-MYSORE



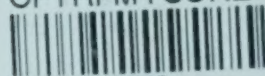
18796

Pascal...

TLMS

PASCAL

CFTRI-MYSORE



18796

Pascal..

CONTENTS

Preface	xi
1. INTRODUCTION	1
1-1 Introduction	1
1-2 Programming a Computer	2
2. <u>SIMPLE PASCAL PROGRAMS</u>	6
2-1 Introduction	6
2-2 Structure of Simple Programs	6
2-3 Output	8
2-4 Syntax and Syntax Diagrams	11
2-5 A Few Fine Points	14
2-6 Exercises	15
3. <u>THE ASSIGNMENT STATEMENT AND ARITHMETIC EXPRESSIONS</u>	17
3-1 Introduction	17
3-2 The Assignment Statement	18
3-3 Arithmetic Expressions	22
3-4 Constants	25
3-5 Syntax Diagrams	26
3-6 Some Notes on Programming Style	29
3-7 Exercises	31

4.	<u>INPUT AND OUTPUT</u>	33
4-1	Introduction	33
4-2	Output with WRITELN	34
4-3	Output with WRITE	40
4-4	Input with READ and READLN	42
4-5	Writing Interactive Programs	46
4-6	Notes on Programming Style and Pitfalls	48
4-7	Exercises	50
5.	<u>INTEGERS AND INTEGER VARIABLES</u>	53
5-1	Introduction	53
5-2	Representing Numbers in a Computer	53
5-3	Type Errors and Type Conversions	56
5-4	Arithmetic Expressions Using INTEGER Values	58
5-5	Writing INTEGER Values	61
5-6	Syntax Diagrams	62
5-7	Exercises	63
6.	<u>LOOPS AND THE FOR STATEMENT</u>	65
6-1	Introduction	65
6-2	The FOR Statement	66
6-3	Compound Statements	69
6-4	Some Subtle Points about FOR Loops	71
6-5	Flowcharts	75
6-6	Syntax	77
6-7	Notes on Programming Style and Pitfalls	78
6-8	Exercises	79
7.	<u>ARRAYS</u>	82
7-1	Introduction	82
7-2	Use of Subscripted Variables	83
7-3	Declaring Arrays	86
7-4	Higher Dimensional Arrays	88
7-5	Nested FOR Loops	89
7-6	Syntax	95
7-7	Remarks on Techniques, Style, and Pitfalls	95
7-8	A Programming Example	99
7-9	Some Notes on Program Debugging	103
7-10	Exercises	105

8. LOOP CONTROL WITH WHILE AND REPEAT	108
8-1 Introduction	108
8-2 The WHILE Statement	108
8-3 BOOLEAN Variables	110
8-4 Boolean Expressions	111
8-5 Relational Operators and Conditions	112
8-6 Reading Unknown Amounts of Data	115
8-7 Loops with REPEAT...UNTIL	118
8-8 Syntax Diagrams	120
8-9 Some Hints on Programming Style and Pitfalls	123
8-10 A Programming Example	126
8-11 Exercises	129
9. THE IF AND CASE STATEMENTS	133
9-1 Introduction	133
9-2 The IF Statement	134
9-3 Flowcharts with Branching	138
9-4 Nesting IF Statements	139
9-5 The CASE Statement	141
9-6 Syntax	144
9-7 Programming Hints: Efficiency and Testing	145
9-8 A Programming Example: Sorting	148
9-9 A Programming Example: Loan Payment Schedule	152
9-10 Exercises	155
10. CHARACTER VARIABLES AND CHARACTER STRINGS	162
10-1 Introduction	162
10-2 Character Variables	163
10-3 Packed Arrays	168
10-4 Character Strings	170
10-5 Syntax	176
10-6 Exercises	177
11. PROCEDURES AND FUNCTIONS	181
11-1 Introduction	181
11-2 Defining and Calling a Procedure	183
11-3 Value Parameters	185
11-4 Variable Parameters	188

11-5	Scope of Variables: Local and Global Variables	191
11-6	Block-Structured Programs	195
11-7	Procedures with a Single Result: Functions	200
11-8	Procedures and Functions as Parameters	202
11-9	Syntax Diagrams	205
11-10	Standard Procedures: Input-Output and Packing Arrays	207
11-11	Standard Functions: Arithmetic and Ordinal Functions	209
11-12	A Programming Pitfall: Side Effects	212
11-13	Programming Hints: Testing and Debugging a Modular Program	214
11-14	Exercises	215

12. RECURSIVE SUBPROGRAMS **219**

12-1	Introduction	219
12-2	A Simple Example	221
12-3	A Practical Example	224
12-4	Forward References	229
12-5	Syntax	231
12-6	Notes on Style and Pitfalls	232
12-7	A Programming Example: Sorting	232
12-8	Exercises	238

13. SIMPLE DATA TYPES **241**

13-1	Introduction	241
13-2	Defining New Simple Types by Enumeration	241
13-3	Ordering of Values	243
13-4	Using Enumerated Types in CASE Statements	245
13-5	Subrange Types	247
13-6	Syntax	249
13-7	A Programming Example: Analyzing Student Grades	249
13-8	Exercises	253

14. SETS AND ARRAYS AS STRUCTURED TYPES **255**

14-1	Introduction	255
14-2	Elementary Set Theory	256
14-3	Sets in Pascal	257
14-4	Set Operations	259
14-5	Arrays of Arrays	263
14-6	Variables of Defined Types as Subprogram Parameters	265
14-7	Syntax	266
14-8	Exercises	266

15.	<u>RECORDS AND FILES</u>	268	ix
			CONTENTS
15-1	Introduction	268	
15-2	Defining Record Types and Declaring Records	269	
15-3	Referencing Records and Their Fields	273	
15-4	Files	277	
15-5	Textfiles	285	
15-6	Syntax	288	
15-7	A Programming Example: Student Records	290	
15-8	Exercises	296	
16.	<u>POINTERS AND LISTS</u>	298	
16-1	Introduction	298	
16-2	Pointers	301	
16-3	Linked Lists	304	
16-4	Syntax	308	
16-5	A Programming Example: Editing Text	309	
16-6	Exercises	314	
	Appendix A	316	
	Appendix B	317	
	Index	327	

PREFACE

This book is intended to be an introduction to programming in the Pascal language. It was written to fill the need for a book that a nontechnically oriented beginner would be able to read—and, we hope, even enjoy reading. We have not aimed at a textbook for majors in computer science; consequently we do not treat topics such as data structures and algorithms in any systematic manner.

One of the features of the book is that the topics are arranged in such a way that the exposition is always sequential, in the sense that understanding never depends on material to be covered later. We hope thereby to encourage students to read carefully as they go and not postpone understanding of some concepts until later. At the same time, the material is arranged so that students can quickly start to write simple programs and actually use the computer as early as the first week of the course.

Examples of complete programs are included at the ends of many of the chapters to expose students to some of the important concepts of program development, structured programming, and flowcharting. We chose to utilize one of the modular styles of flowcharting (essentially Nassi-Schneiderman), since we believe that this is helpful in gaining a good feel for structured programming. We have also waited to introduce flowcharting until after loops are discussed, since we find that students tend to feel that flowcharts are useless if they first see them used in discussions of straight-line code.

Even though this is an elementary book, we decided to present syntax in a formal way by using, with minor alterations, the syntax diagrams from Jensen and Wirth's *Pascal User Manual and Report*. We believe that the clarity and lack of ambiguity of this approach outweigh the small amount of difficulty that some students may initially experience in understanding the

diagrams. Also, the diagrams, once understood, are useful as a quick summary of the syntax for later reference. (Furthermore, a beginner could choose to skip the syntax diagrams in a first reading of the book.)

The proliferation of versions and extensions of the Pascal language poses something of a problem in writing a text on "Pascal." We have omitted discussions of extensions (e.g., predefined character string types) and have tried to describe the language as originally presented in Jensen and Wirth and clarified in the 1979 draft standard [B. W. Ravenal, "Toward a Pascal Standard," *Computer*, Vol. 12, No. 4 (April 1979) pp. 68–83.] In the interests of discouraging sloppy programming, we have, however, avoided any use (and indeed mention) of the GOTO statement and statement labels, although GOTO and LABEL have perforce been included in the list of reserved words in Appendix A. Examples in the text were run on a CDC CYBER 175, using the E.T.H. Zurich/University of Minnesota Pascal 6000 Compiler. We have attempted to make it clear to the reader that Pascal implementations vary, so that some experimentation and consultation of local manuals is advisable.

The first eleven chapters of the book cover those features of Pascal with which one can program tasks like those one would learn how to program in FORTRAN or BASIC. Chapters 12–16 deal with what might be called "advanced" features of the Pascal language—recursion, type definition, structured data types, etc. After completing Chapter 11, the student should have developed a reasonable amount of programming sophistication. In the later chapters, therefore, topics are covered at a somewhat brisker pace, although we hope with equal clarity.

We wish to express our appreciation to the numerous friends, students, and reviewers who read various parts of the manuscript at various stages and provided many helpful comments and suggestions. Special thanks go to Arthur Liestman, Sandra Pritchard, and Andrew Spry, who were responsible for writing and working out solutions to most of the exercises, and to Cindy Robins, who, as a high-school-age "beginner," tested the book for readability and suitability for self-study.

GENEVA G. BELFORD
C. L. LIU

PASCAL



INTRODUCTION

1-1 INTRODUCTION

The electronic computer was invented slightly more than 30 years ago. Seldom in the history of the world has a technology developed at such an explosive rate. By now computers play a role in almost every aspect of our daily lives. Computers are used to record our banking transactions, to check our income tax returns, and to control the inventories of the stores where we shop. Computers are used to control the flight of spacecraft, to regulate the processes in a chemical factory, and to analyze the results of complex physics experiments.

We are now beginning to see computers appearing in our homes. Computers are built into microwave ovens to carry out complex sequences of cooking operations without our having to be on hand to turn the dials. Ultramodern sewing machines have built-in computers to allow a wide choice of fancy stitches without mechanical cams. Many of us have enjoyed so-called video games, which are really *computer* games that use our TV screen for a display. If we're anxious to save on gas, we could buy a computer for our car that tells us, with merely the touch of a button, what gas mileage we are getting at the moment. And many of the readers of this book are wearing computers on their wrists to tell them the time and carrying small computers in their pockets to relieve them of the need to remember how to multiply.

Given this pervasiveness of computers, anyone with any curiosity at all should want to learn something about them. There are many questions one might raise: First, what is a computer? (Clearly the computers inside our watches and those that manage our bank accounts are rather different machines!) How is a computer built? What kinds of jobs can a computer do? If we want a computer to do one of those jobs, what one should we buy—and

how do we go about using that computer to get the job done? It is not possible for us to cover all aspects of computing in this book. Our goal is the limited one of teaching you to use a general-purpose computer, that is, one that can be used to carry out a wide variety of tasks involving computations and data manipulation.

If you are interested in learning all about automobiles, an obvious first step is to learn to drive one. This will give you a good feel for the capabilities and limitations of cars. Similarly, if you learn how to use a computer, you will not only be able to use it to solve various kinds of problems, but you also should gain a reasonably good understanding of what the capabilities and limitations of a computer are. Your view, as a beginner, will necessarily be a bit limited. Just as when you first learn to drive a car you are unlikely to have a feel for what it's like to drive in the Indianapolis 500, so when you first learn to use a computer you may not be able to imagine how you could get the computer to do such complex tasks as maintaining the records of a large corporation or guiding a spacecraft to the moon. However, as you progress through this book, you will find yourself gaining a basic understanding of how computers can help solve real problems.

1-2 PROGRAMMING A COMPUTER

A visitor from Mars has just presented us with a robot which he claims can do all sorts of wonderful things. Two immediate questions arise: What are the wonderful things that the robot can do? How do we instruct the robot to do these wonderful things for us?

Similarly, when you discover the availability of a computer at school or at work, you might ask: What are the wonderful things that a computer can do? How do I instruct the computer to do these wonderful things for me?

This book is mainly concerned with answering these two questions. To do this, we will concentrate on teaching you how to give instructions to a computer, since providing an answer to the second question should go a long way toward answering the first. If you are capable of giving instructions to the computer you must have a reasonably good idea of what the computer can and cannot do. To return to the car analogy, when you learn how to put a car in forward and backward gears, you know that a car can move forward or backward. Not having been taught to move a car sideways, you might feel reasonably safe in concluding that cars do not have that capability—although it might occur to a bright student that making a car hop sideways could just be too tricky a maneuver for a beginning course!

The first thing to get clear in learning to use a computer is that a computer is just a machine. Any instruction given to a computer must be given in such a way that the computer can understand exactly what the instruction is. Indeed, a computer can only follow instructions given in a code known as *machine language*. Most computers are built in such a way that these machine language instructions are in the form of sequences of 0's and 1's. (For example, 0111011010 might mean "multiply 3 times 2.") It is

clear that we ordinary human beings would have some difficulty remembering instructions coded like this. Unfortunately, there are compelling engineering reasons for building computers this way. At first glance it seems that we are in a dilemma; namely, a computer can only understand instructions given in the form of sequences of 0's and 1's, while a human being would find it extremely cumbersome to give instructions in this form. The situation, however, is not an unfamiliar one. If we wish to talk to a Martian in English while the Martian can only understand Martian, we need an interpreter. Indeed, the same idea works for communication with a computer, as Figure 1-1 illustrates.

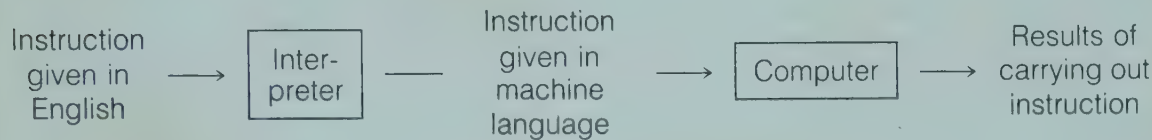


Figure 1-1

The illustration is not accurate in one particular; namely, we really cannot give instructions in plain English to the computer. There are two reasons for this: First, ordinary English is too complicated for us to “train” an interpreter to understand the whole language. (As will be seen, the interpreter is less intelligent than a human.) Second, ordinary English can be ambiguous. There are obvious examples such as, “Find the square root of 4 times 5.” (Is the answer 10 or $2\sqrt{5}$?) But even carefully written instructions can leave something to be guessed at. For example, the instruction “Add to each salesperson’s salary a bonus of 5 percent of the difference between his total sales and his target sales” leaves open the question as to whether there can be a *negative bonus*, that is, a deduction from the regular salary, if total sales are below target sales. To avoid these problems, what has actually happened is that simple, English-like languages have been designed with the following requirements in mind:

1. They should be close enough to English so that they are easy for human beings to learn, understand, and remember.
2. They should be flexible enough so that it is convenient to instruct the computer to carry out quite complicated tasks.
3. They should be precise enough so that there can be no ambiguity as to the intention of the user.
4. They should make translation to machine language not only possible but also reasonably efficient.

Such artificial languages are referred to as *programming languages*. (We should note that a set of instructions given to a computer to define some task is known as a *program*; writing such a set of instructions is known as *programming* the computer.) Since the early 1950s many programming languages have been proposed. The more important and widely used ones include the languages called FORTRAN, PL/I, COBOL, BASIC, ALGOL, APL, and Pascal. It is not possible, nor is it necessary, for us to teach a student all

these languages in one course. The best approach for a beginner is to pick one language and learn to use it proficiently. Since most of these languages bear a strong resemblance to each other, both in appearance and in underlying design philosophy, students should have little difficulty in learning new languages on their own after they have mastered one.

The language we have picked to describe in this book is called *Pascal*. Pascal is a relative newcomer among programming languages, having been designed by Niklaus Wirth (a professor at the Federal Institute of Technology in Zurich, Switzerland) in 1969. Having been built on some 15 years of previous development and experimentation in the design of programming languages, Pascal is in many ways easier to use—and has fewer pitfalls for the unwary—than many of the older languages. It is therefore a good choice as a language for a beginner to learn.

The next question is: How are interpreters trained to translate instructions given in the Pascal language into instructions in machine language for the computer to execute? As we hinted earlier, human interpreters are not used. There are many obvious reasons for this: Humans are not always reliable; they won't work 24 hours a day; they are slow; they might get sick; they take vacations and coffee breaks; and it takes a long time to train them. A much better idea is to build a machine that will act as an interpreter. As a matter of fact, the best candidate for such a machine is the computer itself. When provided with a set of instructions detailing how to carry out the translation, the computer can perform the task of translating instructions given in Pascal into instructions in machine language. Figure 1-2 illustrates the situation.

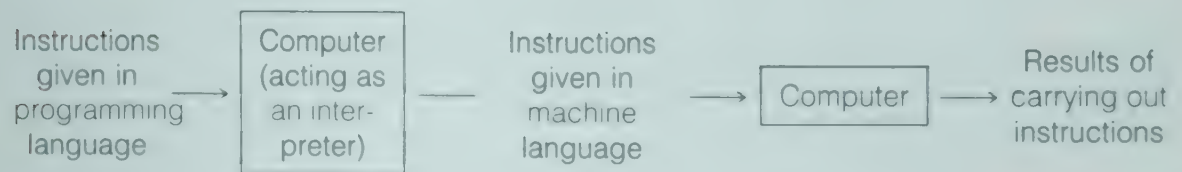


Figure 1-2

The set of instructions (or program) given to a computer to tell it how to translate from an English-like programming language to machine language is usually referred to as a *compiler*. Luckily for those of us who want to use computers, compilers have been written (and are readily available) to translate from almost any popular programming language to the machine language of whatever computer we might have available. Just as human translators may translate the same sentence in slightly different ways, so two different compilers on different computers may translate the same program slightly differently. Programming language features, the detailed effects of which may vary from computer to computer, are referred to as being *implementation-dependent*. Because of this dependence on implementation, we cannot always describe to you in perfect detail what a program written in Pascal will do on your computer. But as we describe the Pascal language, we will try to warn you when your computer may behave slightly differently from ours.

Actually, compilers are only one of a number of programs that automatically come into play to make your use of the computer easy. Another program (sometimes called an *executive* or *monitor*) looks at your job, sees what programming language you have written it in, calls on the proper compiler to translate it, and then sees to it that the resulting machine language instructions are carried out. Still other programs handle the details of fetching the data you want worked on from wherever it may happen to be stored and of returning results to you in the form you ask for—e.g., displayed on a TV screen or printed on paper. On large central computers, such as those maintained at colleges and universities, accounting programs check whether you have money to use the computer and, if so, bill the cost of your job to your account (which, like your bank account, is maintained inside the computer). In short, one no longer should talk about “the computer” but about “the computer system,” where the system includes not only the *hardware* (the electronic machine itself) but also *software* (the set of prewritten programs that make it convenient for you to get your jobs done).

2

SIMPLE PASCAL PROGRAMS

2-1 INTRODUCTION

As we noted in the preceding chapter, a computer program is a set of instructions given to the computer. This book explains how to write programs in the particular language known as Pascal. In this chapter, we shall show you what a Pascal program looks like by introducing one of the simplest (and one of the most important) instructions we can give to the computer.

Instructions given to a computer, like those written in ordinary English, are divided into sentences, which in Pascal are referred to as *statements*. The major part of a Pascal program consists of a number of statements arranged in sequential order. This corresponds to a set of instructions to be followed in that same order. In addition to these statements, which are explicit instructions to the computer to perform particular tasks, the program must also include other information that the computer needs in order to carry out these tasks successfully. The situation is analogous to that of a truck driver ordered to drive a particular truck from Chicago to New York. In addition to explicit instructions on how to get to New York, the trucker might also like to know the truck's weight so that, for example, he or she can avoid state weighing stations if the truck is overweight.

2-2 STRUCTURE OF SIMPLE PROGRAMS

The first information that needs to be given to the compiler in a Pascal program is the program's *name*. Every program must have a name to be used

for identification purposes. A name is made up of characters. The characters may be either letters (A, B,..., Z) or digits (0, 1, 2,..., 9), but the first character must be a letter.) Thus,

REPORT TAX5 Q74J AAA

are all possible names for programs, while

7RATE AB* JOHN DOE

are illegal names for programs. 7RATE is illegal because it begins with a digit. AB* is illegal because it contains an *, which is a character that is neither a letter nor a digit. JOHN DOE is illegal because it contains a blank, which is neither a letter nor a digit but is a character. You can make up names that are as long as is necessary to identify your program descriptively, for example, TAXCOMPUTATION or INVESTMENTUPDATE. But you should be warned that some compilers will only look at the first eight characters of a name, and they will confuse two programs named, for example, TAXCOMPUTATIONFED and TAXCOMPUTATIONSTATE.

There is one more restriction—a name cannot be a *reserved word*. The reserved words are those that have special roles to play in the language, so that it would confuse the compiler if you used them for other purposes. For handy reference we have listed all of the Pascal reserved words in Appendix A.

Every Pascal program begins with a heading that gives its name, e.g.,

PROGRAM PAYROLL ;

Note that the heading begins with the word PROGRAM. PROGRAM is a reserved word indicating that this is the beginning of a program and that the program's name will follow. In this case the program's name is PAYROLL. (Since the word PROGRAM is reserved, PROGRAM PROGRAM is illegal.) The heading is then followed by a program *block*, which, in the case of very simple programs, consists of the list of instructions to be carried out by the computer. This list of instructions, or statements, is prefaced by the word BEGIN and followed by the word END. Thus a simple program looks like this:

PROGRAM SIMPLE ;
BEGIN

(list of statements)

END .

Notice that a period follows the END to indicate the end of the program. (It would appear at first sight that END itself should play this role. But you will see later that ENDS can also appear in the middle of a program.) The other punctuation appearing here is the semicolon, used to separate the heading

from the program block. Semicolons are also used to separate statements from each other, much as they are used in English. We shall see examples of this shortly.

2-3 OUTPUT

Whatever computation we instruct it to carry out, the computer must eventually supply us with the results. Thus, any computer must possess one or more devices through which it can communicate to us the results of the task we have assigned to it. These devices are usually referred to as *output devices*. Perhaps the most common output device is a TV-like screen, a familiar example of which is the display screen for computer games. Another common output device is a *printer*, a computer-controlled, typewriterlike device that the computer uses to supply you with a printed paper copy of your results. Both printers and display screens may be combined with a keyboard, much like that on a typewriter, to form a device called a *terminal* that provides you with two-way communication with the computer. Computers can also use a card punch to output results on the familiar punched cards or a plotter to trace figures and curves on paper. There are even more exotic output devices now at various stages of development: A computer might provide audio output to the extent of giving you your answers in spoken English; it might produce the negatives for the printing of a book; or it might control the movement of an industrial robot on an assembly line.

Consequently, a programming language must have facilities for instructing the computer to output results of computations. In Pascal, the following statement can be used to instruct the computer to print a line of text on the printer:

```
WRITELN( 'EXAMPLE 21' )
```

The general form of the statement is the word `WRITELN`, which is an abbreviation for "write line," with the text to be printed enclosed by single quotation marks (i.e., apostrophes) and then a pair of parentheses. Thus, in the example above, the text

EXAMPLE 21

will be printed. Consider the following program:

```
( PROGRAM PRINT1 ( OUTPUT ) ;  
  BEGIN  
    WRITELN ( 'NAME: J. J. DRAKE' ) ;  
    WRITELN ( 'AGE: FORTY-TWO ( 42 )' ) ;  
    WRITELN ( 'SSN: 010 43 2615' )  
  END .
```

Note that the program heading here contains, in addition to the program name PRINT1, the word OUTPUT in parentheses. This indicates to the compiler that the program will have some results to be printed in a standard way (either on the system printer or returned to your terminal). [On your computer system you may not have to include this information; the compiler may simply assume that, unless otherwise specified, results should be handled in the usual way. Such assumptions of standard choices which relieve you of the need for explicit specification are called *defaults*. You will have to consult your instructor (or your computer manual) about such details for your particular system.]

Clearly, the program named PRINT1 will print three lines on the printer, namely,

```
NAME : J . J . DRAKE
AGE : FORTY-TWO ( 42 )
SSN : 010 43 2615
```

Note that the WRITELN statement will print the text within the quotation marks *exactly* as it appears, including all the blanks. For example, the statements

```
WRITELN ( '   * ' );
WRITELN ( ' * *   * * ' );
WRITELN ( '       * ' )
```

will print the pattern

```
  *
 * *   * *
  *
```

on the printer.

Note that the same pattern will be printed by the following statements:

```
WRITELN ( '       *       ' );
WRITELN ( ' * *   * * ' );
WRITELN ( '       *       ' )
```

That is, it doesn't matter whether you explicitly include the last blanks inside the single quotes or simply leave the rest of the line blank. Often, to make your output more readable, you will want to insert completely blank lines, which you can do by the statement

```
WRITELN ( '   ' )
```

or even more simply by the statement

```
WRITELN
```

Thus the sequence of statements

```
WRITELN( 'EXAMPLE' );
WRITELN;
WRITELN( 'OF BLANK LINE' )
```

will print

EXAMPLE

OF BLANK LINE

In the preceding section, we promised examples of the use of the semicolon between statements. All the sequences of statements above form such examples. But examine the program PRINT1 carefully. Semicolons appear *between* the first and second WRITELN statements, and *between* the second and third. However, none is needed after the third. In Pascal, the semicolon is used to separate one statement from another. But no semicolon is needed to separate the last statement of a program from the word END that terminates the program.

Since semicolons separate statements, it is not necessary to write each statement on a separate line, although we usually do so to make programs more readable.* Thus, we could write a program to print a pattern of asterisks as:

```
PROGRAM STAR(OUTPUT);
BEGIN
    WRITELN( ' * ' ); WRITELN( '** **' );
    WRITELN( ' * ' )
END.
```

You may be thinking that printing out patterns of asterisks doesn't seem too exciting. However, this can be genuinely useful. For example, you might want to put a clearly visible heading on a table of results by prefacing it with the output from

```
WRITELN( ' ***** ' );
WRITELN( ' * * * * * ' );
WRITELN( ' * RESULTS * ' );
WRITELN( ' * * * * * ' );
WRITELN( ' ***** ' )
```

*It is also legal to continue statements from one line to the next, although this is not recommended since it tends to make programs harder to read. Because of the narrow line width, however, examples of statements will run over more than one line in this book. Breaking a WRITELN statement in the middle of a character string is, of course, to be avoided.

Or you might want to print your name or some favorite slogan in giant-sized letters. For example,

```
WRITELN( ' ***      ***      *** ' );  
WRITELN( ' *      *      *      *      * ' );  
WRITELN( ' ***      *      *      *** ' );  
WRITELN( ' *      *      *      *      * ' );  
WRITELN( ' ***      ***      *** ' );
```

2-4 SYNTAX AND SYNTAX DIAGRAMS

The rules for putting together correct sentences in a language are known as the *syntax* of that language. Although you have been required to study the syntax of English, you have probably discovered that you can communicate reasonably well with other persons even if you are sloppy about the rules. Not so with computer programming languages. Even the smallest syntax error will make it impossible for the compiler to understand and translate what you are saying. Even worse, the compiler may misinterpret what you want done if your sloppy knowledge of syntax causes you to write a program that follows the rules but doesn't say what you wanted to say.

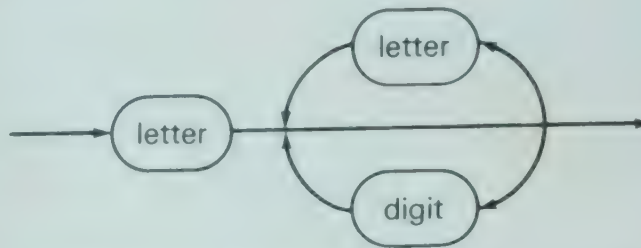
Since syntax must be precisely learned, it must be precisely specified. Look back at our description of the WRITELN statement. You may have had to examine the example before you were quite clear as to where the parentheses should go. We might attempt another specification in English, trying to set down more precisely what a WRITELN statement looks like: A WRITELN statement consists of the word WRITELN, followed by the character (, followed by the character ', followed by the sequence of characters that you want to print, followed by the character ', followed by the character). Besides being tediously verbose, this could still be confusing. For example, at the end of this definition we might wonder whether the right parenthesis is the character being referred to or whether it signifies the end of a parenthetical statement. The ambiguity inherent in English-language descriptions is not suitable for describing the syntax for a programming language. To make the description precise we use a *syntax diagram*—a sort of pictorial representation of what follows what, i.e., of how the various parts of the language must fit together.

Syntax diagrams can probably be best understood by studying some examples. In this section, we will present, as examples, syntax diagrams for the simple language features introduced in this chapter. They will necessarily be incomplete; as we proceed to discuss further details of Pascal, we will introduce the corresponding syntax diagrams. By the time you reach the end of this book you should have an essentially complete picture of the syntax of Pascal. (For reference, we have included the complete syntax of Pascal as Appendix B.)

As a first example, consider the rules we gave for valid program names. Such names belong to the class of Pascal language constructs known as

identifiers. An identifier is a sequence of letters or digits that begins with a letter. The syntax diagram for an identifier is as follows:

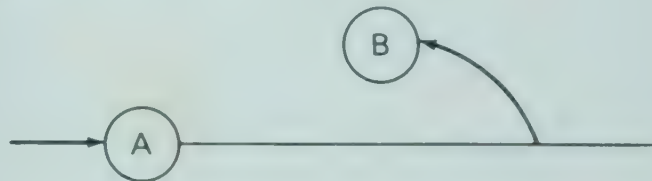
Identifier



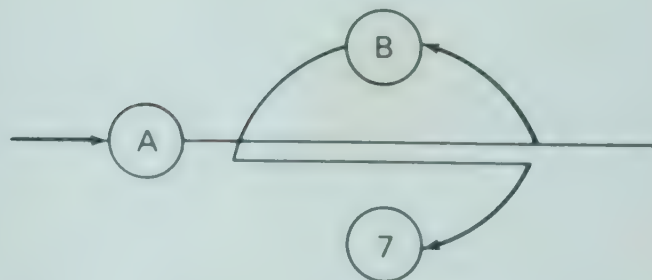
To determine allowable forms for identifiers, we start at the left side of the diagram and follow the arrows through to the right side. Wherever there are branches, we can take any of the possible paths. Carrying out this prescription, we see that identifiers must begin with a letter. After this, what happens? By taking the middle path, we find that a single letter is a valid identifier. But if we wish, we can add another letter, or a digit, and so on. To clarify how this works, let us verify that AB7 is a valid identifier. It begins with a letter:



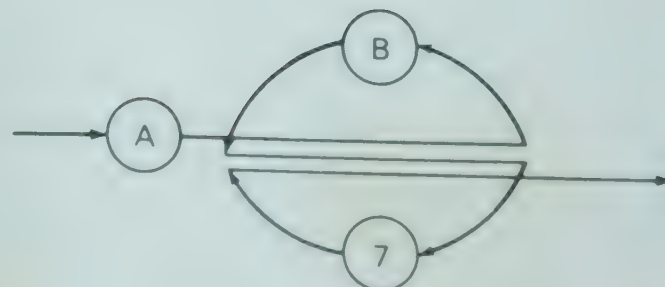
Next, another letter:



Now a digit:



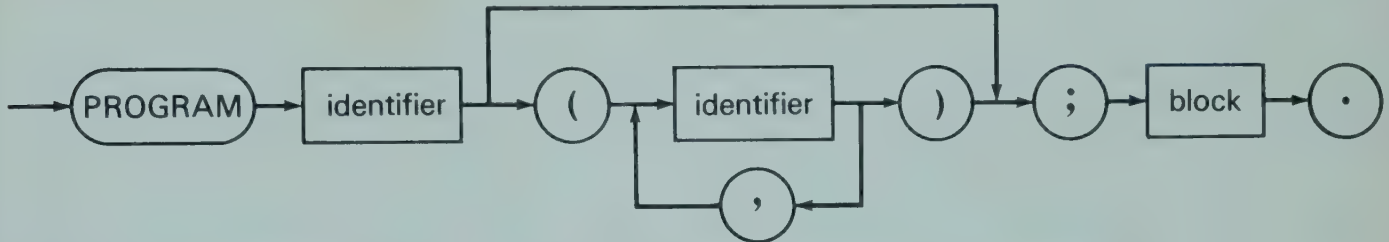
And that's all:



Notice that the syntax diagram doesn't tell you *everything* you need to know about identifiers. You must remember also that only the first eight characters are significant and that an identifier can't be a reserved word (e.g., PROGRAM, BEGIN, END).

The diagram for a *program* is also quite simple:

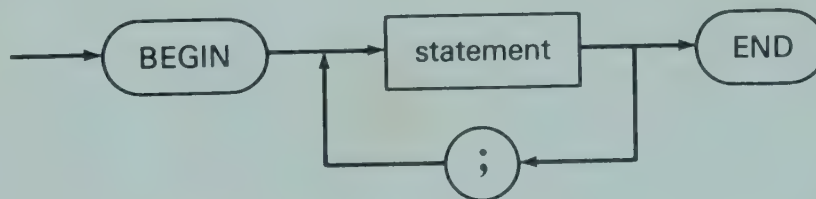
Program



The word PROGRAM appears in capitals here to indicate that it appears as is in the language structure. Identifiers are, of course, as defined in the previous syntax diagram. (The fact that you must look elsewhere for the diagrams for “identifier” and “block” is indicated by the rectangular boxes. Items in ovals or circles must appear exactly as given.) We see that a program begins with the word PROGRAM, followed by an identifier (which we have learned is the program's name), possibly followed by a list of identifiers separated by commas and set in parentheses [e.g., (OUTPUT) to indicate where the results should be sent], followed by a block, followed by a period.

It remains to give a diagram for a *block*. Program blocks can be quite complicated and contain more language features than we wish to discuss now. Blocks consisting of lists of simple statements can be diagrammed as

Block (Simplified)



If you study this diagram carefully, you should have no trouble understanding where the semicolons are needed.

Next, what is the syntax of a statement? There are many different kinds of statements, corresponding to the various types of instructions that one can give the computer. So far we have looked only at statements involving WRITELN—and only at a particularly simple form of those, as you shall see shortly. We will be giving syntax diagrams for the various types of statements as we discuss them. We will leave it as an exercise for you to construct a syntax diagram for the simple WRITELN of this chapter.

One final point: a number of spaces or blanks are in our examples, but nowhere does a space appear in the syntax diagrams. Just as in English, spaces are used in Pascal to separate the “words” of the language. They make

your program more readable but are generally ignored by the computer. For example,

```
WRITELN( 'XYZ' )
```

and

```
WRITELN ( 'XYZ' )
```

are equally valid. The compiler looks for a left parenthesis after `WRITELN` and doesn't care whether there are intervening spaces to skip over. If you should need to write two identifiers in a row, however, a space between them is required, lest the compiler try to interpret them as one long identifier. You will probably be able to recognize other similar situations where spaces are necessary. We shall also point out some places where spaces are forbidden. Even though it requires some extra typing, use spaces liberally to make your program easier to read.

2-5 A FEW FINE POINTS

Careful readers have probably observed the following apparent redundancy: Since the sequence of characters to be printed is enclosed in parentheses, the quotation marks seem to be quite superfluous. This, however, is not the case. The statements

```
WRITELN ( 'MYNAME' )  
WRITELN ( MYNAME )
```

can both be legal, but they will have entirely different meanings. A sequence of characters enclosed in quotes is known technically as a *string* or *character string*. An instruction to print a character string prints that string literally, as you know. `MYNAME` (without the quotes) obeys the syntax rule for an identifier: it is clearly *not* a character string. In Chapter 3 we shall explain what statements like `WRITELN(MYNAME)` mean.

Another interesting question is what to do if you want to use an apostrophe in the line to be printed, e.g., if you want to write the instruction:

```
WRITELN( ' JOE ' S  REPORT ' )
```

When the compiler reads the apostrophe after `JOE`, it will naturally assume that that is the end of the character string. (What kind of confusion the rest of the string will cause depends on your local compiler and the rest of your program. You might try it and see!) What you do is to use two apostrophes:

```
WRITELN( ' JOE ' ' S  REPORT ' )
```

When it reads the second apostrophe after JOE, the compiler will recognize that you don't mean to end the string after JOE, but you do really mean to print an apostrophe. In general, the rule in Pascal is: If a character string is to contain an apostrophe, this apostrophe must be written twice. Then to print the string

```
' THE ' ' END '
```

we use the statement

```
WRITELN( ' ' ' THE ' ' ' ' END ' ' ' )
```

since each of the apostrophes that we want to print out must be written twice. [Note: Be careful not to confuse the familiar double quotation mark (") with the pair of apostrophes (').]

Clearly, you must count the apostrophes carefully to get the printout you want. Suppose you leave out one of the apostrophes after END. Then the computer will not recognize that you mean to end the string and will continue to copy characters to the printer until it finds a single quote. Thus if your program contains

```
WRITELN( ' ' ' THE END ' ' ); WRITELN( ' ' ' )
```

all the characters from the first quote to the last will be considered as a single character string, and the resulting output will be

```
' THE END ' ); WRITELN ( ' ' ' )
```

Notice that it took *two* mistakes to make the program legal; that is, we wrote

```
WRITELN( ' ' ' )
```

which is not a correct WRITELN statement. (Why?)

Now that you are ready to write your own computer programs, let us give you a word of advice. Don't be afraid to do some experimentation. No computer manual ever written answered *all* the questions that might occur to a user. You won't break the computer by trying something that might not work!

2-6 EXERCISES

2-1. Which of the following are valid identifiers?

X2, AB-CD, *IX, IX9, 75, A75, A75.2

2-2. What errors can be found in the following Pascal program?

```
PROGRAM PROBLEM2 ,
BEGIN;
    WRITELN( 'OUTPUT OF' );
    WRITELN( 'QUESTION TWO' )
END
```

- 2-3.** Draw a syntax diagram for the WRITELN statement as described in this chapter. (Since any allowable character may be printed out, use a rectangular box labeled "character" in your syntax diagram.) Do not worry about the complexities we hinted at in Section 2-5, but just diagram the printing of a single character string.
- 2-4.** Write a program to print your first name in giant-sized letters (like the "BOB" at the end of Section 2-3). Be inventive! Experiment with different characters and different sizes.
- 2-5.** Write a program to print your name and address, as for an address label. (Warning: Don't call your program LABEL. This is a reserved word.)
- 2-6.** What will each of the following sequences of WRITELN instructions print? Try them out to check your answers.

- (a) `WRITELN( ' * * * ' );`
`WRITELN( ' * * * ' );`
`WRITELN( ' * * * ' );`
- (b) `WRITELN( ' ' * ' ' ' ' );`
`WRITELN( ' ' * * * ' ' ' );`
`WRITELN( ' * * * * ' );`

SUMMARY OF CHAPTER 2

Every program begins with a heading that contains the program's name. A program name is made up of letters and digits, with the first character being a letter. Reserved words (words that have special meanings in Pascal) cannot be used as program names.

A program that uses an output device must, in some implementations, include the word `OUTPUT` in parentheses following the program name in the heading.

In simple programs the heading is immediately followed by a program block delineated by `BEGIN` and `END`.

The `WRITELN` statement can be used to print a line on the printer or to display a line on the terminal screen.



THE ASSIGNMENT STATEMENT AND ARITHMETIC EXPRESSIONS

3-1 INTRODUCTION

You probably already know that a computer can do arithmetic—addition, subtraction, multiplication, division—at incredible speeds. (Indeed, a modern computer can easily carry out a million or more arithmetic operations within one second.) Obviously, any programming language must have ways to instruct the computer to carry out these arithmetic operations. Suppose we want to tell the computer to add the two numbers 7.5 and 2.27. An obvious and natural way is to write

$$7.5 + 2.27$$

This gets the addition done, but what happens to the sum? It might, of course, be printed as output. However, what if we need the sum later for another computation? In many computational tasks there are “intermediate results” that need to be saved for later use. For example, suppose we are given the grades of the students in a programming course and we wish to compute the averages for seniors, juniors, sophomores, and freshmen so that we can determine which of these groups is doing best. We need to compute the averages of the groups, save them, and later compare the averages to pick out the highest. Fortunately, a computer has a *memory* to provide room for storing intermediate results.

This notion should not be unfamiliar to you. Your pocket calculator almost surely contains one memory location (perhaps more, depending upon how expensive it was) in which you can store a number for later use. In using your pocket calculator, you generally punch in numbers one at a time as you need them for the calculation.

When using a large computer, we often find it convenient to load the computer memory with all the data to be worked on before beginning the actual computation. Let us look more closely at why we want to be able to do this. Suppose, for example, that we know the annual incomes of 10,000 people, and we wish to add them together to find the total income. Now the computer can do this task in a fraction of a second *if* the numbers to be added are readily available. To provide these numbers to the computer, you could type them on a keyboard, as you do with your pocket calculator. Each number would be added to the sum as it is entered. Using this method, it would take hours to add the 10,000 numbers. Meanwhile, the computer itself is mostly idle, waiting for you to enter the next number. You might then think of punching the numbers on cards, say, one number per card. If you have watched a card reader in action, you may think it pretty fast. However, card readers take more than 10,000 times as long to read a card as the computer takes to add two numbers (at a million adds per second). Although the job seems to get done fast, the computer is still spending most of its time waiting for the next number to be read in. Making a computer sit idle in this way is wasteful. So the standard procedure is to read data into the computer *before* it is needed. The numbers to be added are therefore stored in the memory, ready to be fetched and added almost instantaneously.

The computer memory is used to store the instructions to be executed as well as data to be operated on. To see why we need to store the instructions, let us push this example a little further. Suppose we want to compute not only the overall total of the 10,000 salaries, but also subtotals for 20 different groups of workers—secretaries, clerks, managers, computer programmers, etc. Notice that now the data must include job classification information attached to each salary, so that the computer can decide which numbers get added to which totals. The instructions we must give the computer for this task are quite complicated. Memory space for the various subtotals must be set up, and the computer must examine the job classification before adding the income to the appropriate total. This task can still be handled by a computer in less than a second, *if* the data are available *and if* we can feed the computer its instructions fast enough. Again, there's only one way to accomplish this, and that's to have the instructions, as well as the data, stored in the computer's memory for instant access. Probably more than anything else, it was the brilliant idea of storing the program, as well as the data, in the computer that initiated the "computer revolution."

3-2 THE ASSIGNMENT STATEMENT

The main memory of a large computer contains hundreds of thousands of *locations*. You can think of a memory location as a pigeonhole in which a number can be stored. The number stored in such a pigeonhole can be read whenever it is needed. Thus we can instruct the computer to compute

$$7.5 + 2.27$$

and save the result in one location, to compute

$$77.9 - 48.0$$

and save the result in another location, and finally to divide the first result by the second. To carry out such a computation, we need to be able to identify the locations we use to store data so that we can find and read the data later on. We therefore give names to locations. Just like the name of a program, the name of a memory location is an identifier, i.e., a sequence of letters and digits, the first being a letter. (And don't forget that you can't use reserved words and that identifiers more than eight characters long may cause problems.)

The fact that a particular memory location named *X*, for example, contains the number 10.5 is usually expressed by saying that the *variable X* has the *value* 10.5. Variables play roughly the same role in a programming language that they do in ordinary algebra. In algebra, you can give different values to variables, as when you write, "Let $X = 5$." You can also write general formulas (*arithmetic expressions*) using the variable names and then use the same formula for any particular values that you want to carry out the computation for. For example, you know that if $X = 5$ (i.e., the variable *X* is given the value 5) then the expression $X^2 + 22$ evaluates to 47. Similarly, you will learn how to write arithmetic expressions in Pascal and how to instruct the computer to compute their values.

But first you need to learn how to put a number into a memory location. One way is by writing an *assignment statement*. For example, the statement

COST := 5.5

puts the number 5.5 into a memory location named *COST*; alternatively, we say that this statement *assigns* the *value* 5.5 to the *variable* *COST*. (Note the similarity to writing $X = 5$ in algebra.) Physically, what happens is that the assigned value is stored in a memory location that is identified by that particular variable name. It is sometimes useful to think of a variable as the name of a memory location where a data value may be stored.

Consider, for example, the computation described in the beginning of this section. A program to do this might include the statements

```
SUM  := 7.5 + 2.27;
DIFF := 77.9 - 48.0;
QUOT := SUM/DIFF
```

Execution of the first statement computes the sum of 7.5 and 2.27 and stores the result (9.77) in the memory location named *SUM*. The second statement computes the difference of 77.7 and 48.0 and then stores the result (29.7) in *DIFF*. The third computes the quotient of the value (9.77) stored in *SUM* divided by the value (29.7) stored in *DIFF*, and then stores this quotient in a memory location called *QUOT*. [Note that we have to use the diagonal slash (/)

to indicate division in computer languages, since the more familiar horizontal bar is incompatible with the all-in-one-line format of computer instructions.]

To make sure you understand how assignment statements work, let us consider a few more examples. Suppose we write

```
SUM      := 7.5 + 2.2;
INCRE    := SUM + 0.08;
SUM      := INCRE + 2.0;
DECRE    := SUM - 1.5
```

The value of SUM is computed as 9.7, and the value of INCRE is computed as $9.7 + 0.08$, or 9.78. The value of SUM is then *changed* to 11.78, and the value of DECRE is computed as $11.78 - 1.5$. To logically figure out the effect of a sequence of assignments, remember that instructions are carried out one after the other. Once a new value has been assigned to a variable, in this case SUM, the old value is gone forever. Next consider

```
SUM      := 4.0 + 2.5;
DIFF     := 4.0 - 2.5;
SUM      := DIFF
```

After the value of SUM is computed to be 6.5 and the value of DIFF is computed to be 1.5, the value of SUM is set to be that of DIFF. The values of both SUM and DIFF therefore become 1.5.

Incidentally, notice how numbers are written in these examples. In Pascal, any number with a decimal point *must* contain at least one digit both before and after the decimal point. Thus we must write 0.08 (*not* .08) and 4.0 (*not* 4.).

Finally, suppose a program contains

```
SUM      := 4.0 + 2.5;
SUM      := SUM + 0.5
```

The value of SUM is computed as 6.5 in the first statement. In the second statement the value of $SUM + 0.5$ is computed as $6.5 + 0.5$, and SUM is then set to be 7.0. If you understand this example clearly, you've gotten a good grasp on assignment statements. Remember: *First* the expression on the right side is computed, or evaluated. *Then* the result is stored in the memory location identified by the name given on the left side.

As a more interesting example, suppose we have computed the values of two variables, SUM and DIFF, and we want to exchange the two values. If you have been paying attention, you should be able to figure out (we'll explain later, but give it a try yourself first) why the following statements will *not* serve that purpose:

```
DIFF     := SUM;
SUM      := DIFF
```

```
TEMPORARY := SUM;  
SUM      := DIFF;  
DIFF     := TEMPORARY
```

This example reinforces a point that we explained earlier; namely, a location is used to hold a number. In particular, a location can hold one and only one number at one time. Therefore, in the example

```
DIFF := SUM;  
SUM  := DIFF
```

after the execution of the statement

```
DIFF := SUM
```

the number that is in SUM has been copied into DIFF. Thus, after the first statement, both locations are holding the same number, namely, the original value of SUM. Meanwhile, the original value of DIFF has been destroyed. Consequently, the second statement

```
SUM := DIFF
```

is a redundant statement that does not change anything. To achieve the goal of exchanging the two numbers in SUM and DIFF, we need to use a third location, which we have called TEMPORARY, to hold the number in SUM. This allows us to put a new value in SUM while saving the original value of SUM in TEMPORARY. (If you have difficulty understanding how this might work, consider the following situation: Suppose John's cup is filled with coffee and Mary's cup is filled with tea, and we want to exchange the contents of the two cups. Obviously, we need to empty the coffee in John's cup into a third cup temporarily, pour Mary's tea into John's cup, and then pour the coffee in the third cup into Mary's cup.)

We have not yet discussed how a programmer gives names to memory locations. Actually, the compiler takes care of this for you, but you must tell the compiler what variable names you will be using in your program so that the compiler will know to set aside a certain number of memory locations, with the specified names attached, for your use. To give the compiler this information, you list the variable names that you want to use at the beginning of your program block. A complete program to exchange two values might read:

EXAMPLE 3-1

```
PROGRAM EXCHANGE (OUTPUT);  
  VAR SUM, DIFF, TEMPORARY : REAL;
```

```
BEGIN
    SUM  := 4.2 + 1.7;
    DIFF := 4.2 - 1.7;
    TEMPORARY := SUM;
    SUM  := DIFF;
    DIFF := TEMPORARY;
    WRITELN (SUM, DIFF)
END.
```

When the compiler reads the word `VAR`, it understands that what follows is a list of the variables that you will be using in your program. It then takes care of finding memory locations and, in effect, attaching the names to them for later reference. Notice that after the list of variables there is a colon, followed by the word `REAL`. `REAL` is the *type* of the variable, i.e., the kind of information to be stored in the location. We will be discussing variable types and what `REAL` means later. At this point, just note that numerical values can be assigned to variables of type `REAL`.

There is another feature of this short program that you should notice. The last statement is

```
WRITELN (SUM, DIFF)
```

Notice the absence of quotation marks. What will be printed is *not* the character strings `SUM`, `DIFF` but the *values of the variables* `SUM` and `DIFF`. We will be looking closely at this feature in the following chapter.

3-3 ARITHMETIC EXPRESSIONS

In the last section we introduced arithmetic expressions informally. Here we will give you a more complete story on how to instruct the computer to carry out arithmetic operations. In Pascal, the following symbols are used to denote the arithmetic operations:

- + addition
- subtraction
- * multiplication
- / division

For example, the meanings of the following statements should be obvious.

```
TOTAL := 750.0 + 52.5
NET    := GROSS - EXPENSES
INTEREST := 0.08 * PRINCIPAL
DIAMETER := CIRCUMFERENCE / 3.1416
```


These operators are referred to as *binary operators*; that is, they operate on *two* operands to give a single result. The operands, as you see, are written one on either side of the operators. Thus

$$A + B$$

denotes the operation of addition applied to the operands A and B.

As in ordinary algebra, the subtraction symbol (−) is also used to denote negative numbers. Thus

$$X := -5.4$$

means to store −5.4 in the location named X. Here the operator denoted by − is applied to a single operand. It is therefore usually referred to as the *unary minus* in this context to distinguish it from the binary subtraction operator. (There is also a unary plus, not used much since it is not necessary.)

The differences between the way arithmetic expressions are written in ordinary algebra and the way they are written in Pascal are occasioned by two problems: Everything must be written on one line (hence X/Y instead of $\frac{X}{Y}$) and names can be longer than one character (hence multiplication signs cannot be omitted and we must write $X * Y$ instead of XY).

Expressions can, of course, be very complicated combinations of operators, variables, and constants. In order to specify which operands go with which operators, you can use parentheses in Pascal, just as you can in ordinary algebra. Thus

$$(A+B) * (C+D)$$

means that the sums $A+B$ and $C+D$ are first computed, following which these two quantities are multiplied together. That is, A and B are the operands for the first addition, and C and D are the operands for the second. The operands for the multiplication are then the quantities $(A+B)$ and $(C+D)$.

Now consider the following example:

$$(A + (B * C)) + D$$

Here B and C are the operands for $*$, A and $(B * C)$ are the operands for the first $+$, and $(A + (B * C))$ and D for the second $+$.

You may be thinking that it could become tedious to write parentheses for every operation. Indeed, if you were to write the expression of the previous example as a formula the way you did in high school algebra, you would probably omit all parentheses and simply write

$$A + BC + D$$

or, writing the multiplication operator,

$$A + B \times C + D$$

Pascal in fact allows the analogous expression, i.e.,

$$A + B * C + D$$

In order to interpret which operands go with which operators in such an expression, some rules are needed. In high school you learned that in the absence of parentheses, multiplication and division are to be carried out before addition and subtraction. Similarly, the Pascal compiler applies rules known as *rules of precedence* to evaluate arithmetic expressions. The rules of precedence specify in what order operations are to be carried out in the absence of parentheses. The order of carrying out arithmetic operations in Pascal is (1) multiplication and division and then (2) addition and subtraction. This means that multiplications and divisions are carried out before additions and subtractions, just as in algebra. Thus in evaluating the Pascal expression

$$A * X + Y$$

A and X are first multiplied together, then Y is added. Some expressions *require* parentheses. For example, in $(A + B) * (C + D)$ we need the parentheses to indicate that (contrary to the "multiplication before addition" rule) we want the additions to be carried out first.

Sometimes there is apparent ambiguity not resolvable by an appeal to operator precedence. Consider the legal Pascal expression

$$X/Y * Z$$

Is this equivalent to $(X/Y)*Z$ or to $X(Y/Z)$? The Pascal rules of precedence include a rule that says that a sequence of operations of equal precedence is carried out from left to right. That is, any sequence of multiplications and divisions (or any sequence of additions and subtractions) is carried out from left to right. Thus $X/Y * Z$ is equivalent to $(X/Y)*Z$. For example, $12.0 / 2.0 * 3.0$ has the value 18.0 (not 2.0). As another example, $X/Y/Z$ is equivalent to $(X/Y)/Z$.

The order in which arithmetic operations on REAL variables are carried out may be summarized as follows:

1. Expressions in parentheses are evaluated first.
2. Multiplication and divisions are carried out before additions and subtractions.
3. In case of equal precedence, operations are carried out from left to right.

There might seem to be further problems with expressions like

$$12.0 / -2.0 * 3.0$$

But these are eliminated by a rule that *no two operators may be adjacent* in a Pascal expression. Thus this expression would have to be written

$$12.0 / (-2.0) * 3.0$$

(operators separated by parenthesis).

Let us go slowly through a fairly complicated example of expression evaluation to see how all this works. Consider the expression

$$72.0 / (2.7 / 0.3) * (1.2 + 2.8) + 5.5 / 1.1$$

The factor $2.7/0.3$ evaluates to 9.0; similarly, $1.2 + 2.8$ evaluates to 4.0. Thus, after evaluating the expressions in parentheses, the given expression becomes

$$72.0 / 9.0 * 4.0 + 5.5 / 1.1$$

Now we do the multiplications and divisions, carrying out each sequence from left to right. This gives

$$32.0 + 5.0$$

The last step is to do the addition to obtain a value of 37.0.

A final note: Don't hesitate to put extra parentheses in an expression if it helps to make clearer (to you and to any other reader) what it means.

3-4 CONSTANTS

Suppose we want to write a program that does a lot of calculations on circles, spheres, etc.—all involving the familiar constant π . Writing out the number 3.14159265 every time is not only tedious but is also liable to typing errors. So we might declare a variable named PI and include the statement

$$PI := 3.14159265$$

at the beginning of the program. From that point on, every time we want to use the constant 3.14159265 we can use the variable PI instead. This approach is convenient but has some drawbacks from the point of view of keeping our program tidy. Defining PI to be a variable implies that we might change its value during the course of computation. To make it impossible to change the value of PI, we can define PI to be a *constant* instead of a variable.

A constant is a name or identifier to which a value is assigned at the beginning of a program. Like variables, constants can be thought of as named memory locations in which data values are stored. But the values of constants are fixed; you cannot change the value of a constant by a statement in your program.

To define PI as a constant we write

```
CONST PI = 3.14159265;
```

after the program heading. (See Example 3-2; notice that constants are declared *before* variables.) The compiler then automatically puts the number 3.14159265 in a memory location named PI. We can then refer to PI wherever we want to in our program.

EXAMPLE 3-2

(Program to print the volume of a sphere of radius 40.0)

```
PROGRAM SPHERE (OUTPUT);  
  CONST PI = 3.14159265;  
  VAR RADIUS, VOLUME : REAL;  
BEGIN  
  RADIUS := 40.0;  
  VOLUME := (4.0/3.0) * PI * RADIUS *  
    RADIUS * RADIUS;  
  WRITELN (VOLUME)  
END.
```

As another example of use of constants, suppose a program computes investment returns on the basis of two fixed interest rates. The rates are most conveniently defined as constants:

```
CONST RATE1 = 0.092;  
      RATE2 = 0.105;
```

Then if the interest rates drop and the rates in this program must be changed, all we have to do is to change these definitions, which will be easy to find in their location right after the program header. We will not have to search down through the program and change assignment statements, as we would if the rates were declared as variables.

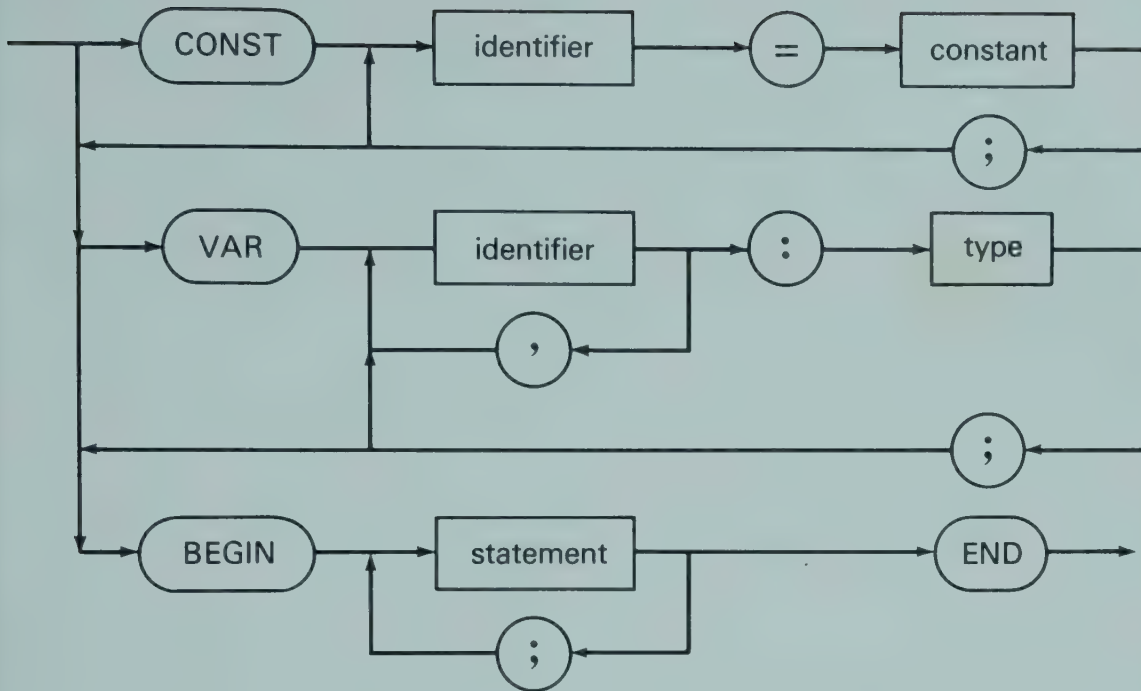
Incidentally, when you write

```
SUM := 5.2 + 2.8
```

the numbers 5.2 and 2.8 are also constants, stored in the computer's memory just like PI or RATE. They just aren't given extra, easy-to-remember names, but are referred to as "5.2" and "2.8."

3-5 SYNTAX DIAGRAMS

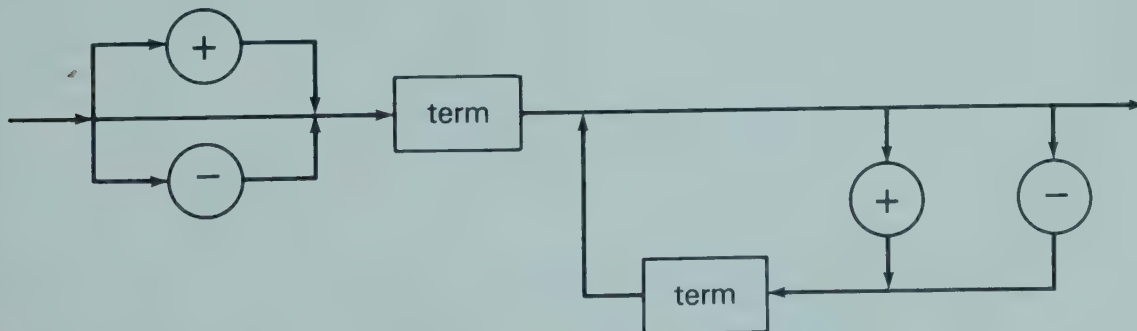
First, let us expand our definition of a *block* to include declarations of variables and constants.



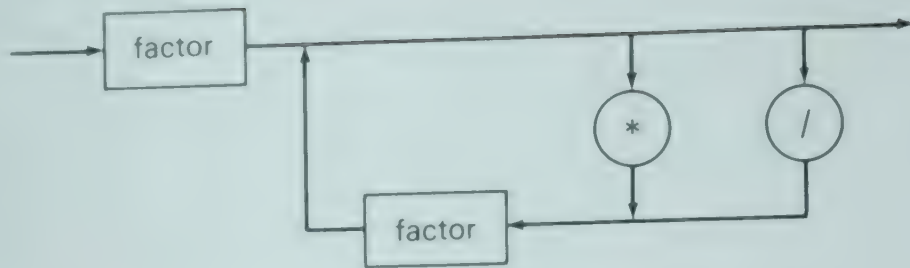
Notice the overall structure of this diagram. We can declare variables or constants or both (or neither)—but the constants must precede the variables, and all declarations must precede the set of statements to be executed.

The complete Pascal language syntax includes a formal definition for an *expression*, which is a more general construct than the simple arithmetic expressions involving REAL variables and constants that are discussed in this chapter. Looking at the syntax for this limited case, we see that (as in ordinary algebra) an expression is a sum of *terms*, which are in turn products of *factors*. Thus we can diagram a *simple arithmetic expression* as

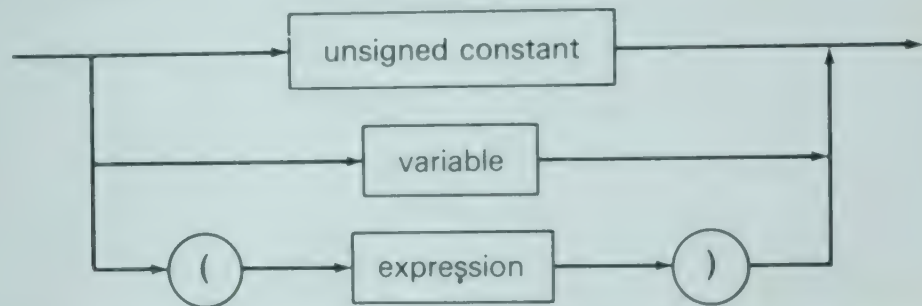
Simple Arithmetic Expression



You should have no problem following this. An expression begins with an optional sign, followed by a term. This term may be followed by a **+** or **-** sign, followed by another term, and so forth. The diagram for *term* (as used in this chapter) is equally easy:

Term (Simplified)

Finally, what is a *factor*? Again, we present a simplified version. [Note that factors cannot be preceded by + or - signs (hence “unsigned constant”), since to do so would cause confusion with addition and subtraction.]

Factor (Simplified)

We appear to have a circular definition! We have defined the concept of *expression* by using *term*, *term* by using *factor*, and now *factor* by using *expression*. However, such definitions make perfect sense in this context, and you will see many examples throughout this book. All this says is, as we said verbally, that expressions inside parentheses are evaluated *first*. Such expressions may involve other expressions inside parentheses, and so forth—but *not forever*! This is the critical feature to notice—eventually we get to the point where the factors involve only constants and variables. (For practice, work through the arithmetic expressions in Section 3-3 to see how the syntax diagram is followed.) Definitions of this sort—which appear to be circular but really aren’t, since eventually you escape from the circle—are called *recursive*.

To complete the syntax diagrams for this chapter, we need one more—a diagram for the *assignment statement*:

Assignment Statement

Notice that we have included the two characters that make up the assignment symbol ($\text{:}=\text{}$) together in one circle. This means that the pair is to be treated as a single entity. In particular, it is an error to separate the colon from the equals by a space or blank.

3-6 SOME NOTES ON PROGRAMMING STYLE

3-6-1 Naming Variables

One of the first things you have to do in writing a program is to determine what your variables are—both data and answers. And then you have to choose names for them. This seems to be a trivial task, but the choice you make can make all the difference between a cryptic program that not even you can understand next week and a program that even a nonprogrammer can glance at and get an idea of what's happening. For example, it is obvious that the statement

$$\text{INTEREST} := \text{PRINCIPAL} * \text{RATE}$$

makes more sense than the statement

$$Z := X * Y$$

Even worse than using neutral letters is choosing bizarre, irrelevant names. We have, alas, seen programs where all the variables were girls' names. Imagine trying to follow a program where all the arithmetic looks like

$$\text{CAROLYN} := \text{CINDY} * \text{CATHY} + \text{SUE}$$

Here's a statement that doesn't look too bad:

$$A := PI * R * R$$

but

$$\text{AREA} := PI * \text{RADIUS} * \text{RADIUS}$$

is much better. Now suppose you want to compute the area of two different circles. Remembering that you can include some digits in names, you might write:

$$\begin{aligned}\text{AREA1} &:= PI * \text{RADIUS1} * \text{RADIUS1}; \\ \text{AREA2} &:= PI * \text{RADIUS2} * \text{RADIUS2}\end{aligned}$$

The moral is: *Choose names that are as descriptive as possible.*

Programmers have come up with other words of advice on choosing names after years of unhappy experiences. For example: *Choose names that you can spell.* (PRINCIPAL is a bad choice if you're likely to misspell it as PRINCIPLE half the time; you might compromise on PRINCIP.) As another example: *Choose names that are different enough so that you don't readily confuse them.* Suppose you are writing a program involving a price and the proceeds of a sale. Don't call your variables PRICE and PROCE! Similarly,

don't use names which differ from each other by interchanging two adjacent letters, like BRKPNT and BKRPNNT. It is too easy to write the wrong name in the wrong place, and then spend hours trying to figure out why the results don't make sense. Names can be hopelessly confused with each other when they differ only in characters that look alike, e.g., the letter O and the digit 0. You are asking for trouble if you call one variable NO and another NO! As another hint to avoid confusion, *don't give your variables misleading names*. In particular, don't assign names that would normally mean something else, such as

```
SIX := 5.0;  
PI  := 6.38
```

3-6-2 Including Comments in Your Program

Notice that we prefaced Example 3-2 by a statement explaining what the program does. It would be far preferable to include such explanatory material (or *comments*) within the program text itself, so that whenever we print out the program, the comments will also be printed. This way we do not need to worry that next week or next month we may forget what a program is supposed to do and have to painstakingly work it out by following through the statements one by one.

The problem is that the compiler must be able to distinguish comments (which we want it to ignore) from the statements it is to interpret. In Pascal, comments are enclosed in curly brackets:

```
{ THIS IS A COMMENT. }
```

In some computer systems, instead of curly brackets a combination of asterisks and parentheses is used:

```
( * THIS IS A COMMENT * )
```

The combinations (* and *) are treated as single entities; do not separate them by a space. But for better readability, separate the comment itself from the brackets, as we have done here. Since comments, like blank spaces, are ignored by the compiler, a comment may be inserted into your program at any point where it is legal to put a space.

A lengthy comment is usually inserted at the beginning of every program to give an explanation of what the program does. Unless the name of a variable is completely self-explanatory, the role of each variable in a computation should be explained by a comment. Furthermore, a long program should have many comments inserted in it to explain what its various segments (or parts) do. "Commenting" a program (i.e., inserting comments in it) is something of an art. The trick is to include enough comments to help the reader understand the program, but not so many that the reader has trouble finding

the program amidst the comments. We will not always be careful to include comments in the examples in this book, since the surrounding text usually provides a more than adequate explanation of what the example program does. But here is Example 3-2 rewritten to include comments.

```
PROGRAM SPHERE (OUTPUT);
  (* COMPUTES THE VOLUME OF A SPHERE OF
    RADIUS 40.0 *)
  CONST PI = 3.14159265;
  VAR RADIUS (* OF SPHERE *),
      VOLUME (* OF SPHERE *) : REAL;
BEGIN
  RADIUS := 40.0;
  (* NEXT COMPUTE THE VOLUME *)
  VOLUME := (4.0/3.0) * PI * RADIUS *
    RADIUS * RADIUS;
  WRITELN (VOLUME)
END.
```

3-7 EXERCISES

3-1. What value would be assigned to the variable VALUE by each of the following statements?

- (a) VALUE := 7.1 + 9.3 * 1.4
- (b) VALUE := 3.6 + 4.1 - 2.5 * 2.0
- (c) VALUE := 2.0 * 3.6 - 1.4 * 3.1
- (d) VALUE := 4.6/2.0 * 3.2 - 1.6
- (e) VALUE := 4.6/(2.3*2.0)/ 2.0 - 1.5
- (f) VALUE := (1.6+4.91)/3.1 * 2.0

3-2. What value is assigned to the variable VALUE after the following assignment statements have been executed?

```
FIRST := 4.1;
SECOND := 1.3;
VALUE := (FIRST + SECOND)/2.7 - FIRST;
VALUE := VALUE * SECOND
```

3-3. Convert the following algebraic expressions to their Pascal equivalents.

(a) $a + \frac{bc}{d} + ef$

(b) $\frac{a + \frac{b}{6}}{c + dx}$

$$(c) \ a + \frac{b}{c + \frac{d}{e}}$$

$$(d) \ -a + b^2 - d$$

$$(e) \ ax^2 + bx + c$$

3-4. What variable names would you choose for the following values?

- (a) Student identification number
- (b) Social security number
- (c) The number of students in a class
- (d) The number of students in a computer science class
- (e) The width of the classroom
- (f) The age of your pet fish Eric

3-5. Which of the following are not legal Pascal expressions? Explain briefly.

- (a) $(A+B)/C * (D-E)$
- (b) $A + B/C) - D$
- (c) $A + 6.4 \ 3.2$
- (d) $4.6 + -3.5$
- (e) $A * -B/4.13$
- (f) $(A) + ((B) - C) * E$
- (g) $A + \qquad \qquad \qquad B$

3-6. In Section 3-2 the program EXCHANGE was given (Example 3-1) which interchanged the values of two variables SUM and DIFF. Write a program which will interchange three values—FIRST, SECOND, and THIRD—so that the value of FIRST goes into SECOND, the old value of SECOND goes into THIRD, and the old value of THIRD goes into FIRST. Print the values before and after the interchange.

3-7. Write a program to print the average of three values: FIRST, SECOND, and THIRD.

SUMMARY OF CHAPTER 3

An assignment statement has the form

VARIABLE := EXPRESSION

It is used to set the value of the variable at the left-hand side to the value of the arithmetic expression at the right-hand side.

An arithmetic expression is evaluated according to the rules of precedence, that is, multiplication and division are carried out before addition and subtraction.

Variables to be used in a program must be declared before the program block. A variable that can have any real number as its value is declared to be of type REAL. A variable name can be any identifier; that is, variable names follow the same rules as program names.

A name may be attached to a constant value by a CONST declaration. Constants must be declared before variables.

4

INPUT AND OUTPUT

4-1 INTRODUCTION

We have already pointed out the need for communication between humans and computers. To have a computation carried out, we need to supply the computer with a program and the data to be worked on. After completing a computation, the computer needs to return the results to us. We have already discussed (Section 2-3) some of the various types of output devices that the computer can use to return results. *Input devices*—that is, devices by means of which we may provide information (such as a program to be executed) to a computer—can be equally varied. One such device is a *card reader*. We prepare the information by punching it onto cards, and the card reader reads the information (coded by patterns of holes) and transfers it into the computer. There are also devices, popularly called *terminals*, with a typewriterlike keyboard and a display (either a printer or something like a TV screen) so that we can see what we're typing.

If you have done any of the programming exercises of the preceding chapters, you will have already become familiar with one of these two devices. You either had to type your program onto punched cards for input to the computer via a card reader, or you sat at a terminal typing your program line by line, using the computer's editing facility to fix typing errors, and finally submitting the program to the computer to be compiled and run.

In addition to card readers and terminals, there are other, more exotic modes of information entry. For example, there are now computers that can receive spoken English (with an extremely limited vocabulary) as input. "Touch panels" (you may have seen them on microwave ovens) display vari-

ous possible commands or data values that you can choose among by touching the one you want. Optical scanners at your supermarket checkout counter read product codes into the computer, which then automatically computes your bill (from prices stored in its memory) and adjusts the store's inventory records. (Such optical scanners might allow any sort of pictorial data to be input, of course.) Finally, electrical pulses—representing perhaps the output from a physics experiment or from a monitoring device on an assembly line—can be fed directly into a computer system for analysis.

In any programming language there must be facilities to control the input and output of information. In Chapter 2 we explained how to produce lines of output to be printed or displayed on your terminal screen. This allowed you to write simple programs and gain some familiarity with the computer. In this chapter we will give you a more detailed discussion of printed output, as well as explain how you can instruct the computer to read data values from punched cards or from lines you type at a terminal. Unfortunately, more exotic forms of input and output require special hardware and software facilities and are outside the scope of a course on the Pascal language.

4-2 OUTPUT WITH WRITELN

When instructing a computer to output information, we need to specify three things: namely, what output device we wish to use, what results we wish to output, and the format (or appearance) of the output. If you have been using a card reader as input device, you will probably have found that the output of your job is automatically sent to the line printer for printing. Similarly, if you have been working at a terminal, you have probably found that the output is returned to your terminal for printing or display. Thus, depending on your mode of operation, your computer system has a default output device. Only if you want to do something different—e.g., if your terminal has a screen display and you want printed paper output—need you specify an output device.

As you have already learned in previous chapters, the `WRITELN` statement can be used to specify a line of output on the default output device. You have already seen that, for example, the statement

```
WRITELN( 'X' )
```

prints the line

X

(i.e., the characters in quotes are printed verbatim), while

```
WRITELN(X)
```


prints the *value* of the variable X. You have seen that you can print the values of several variables on one line. For example,

```
WRITELN(SUM,DIFF)
```

will print the value of SUM, followed by the value of DIFF. Notice that the names of the variables are separated by commas. In general, a WRITELN statement consists of the word WRITELN followed by a list of items to be printed, the entire list being enclosed in parentheses. Items in the list are separated by commas, just as items in a list would be separated by commas in English. An “item” to be printed might be either a variable value or a character string. Both kinds of items can be included in a single WRITELN statement. For example, a tax-computation program might end with

```
WRITELN('TAX OWED IS',TAX)
```

This statement will print “TAX OWED IS”, followed by the value of the variable TAX, all on one line. Notice that this facility allows you to clearly identify the results of a computation—a highly recommended procedure!

In addition to printing values of variables, we can also print values of expressions. Suppose, for example, that we want to compute and print the sum of three variables, A, B, and C. One way would be to declare a fourth variable, TOTAL, and write an assignment statement

```
TOTAL := A + B + C
```

followed by a WRITELN statement:

```
WRITELN('TOTAL IS',TOTAL)
```

Alternatively, we can avoid declaring and using the variable TOTAL by using an expression in the list of items to be printed:

```
WRITELN('TOTAL IS', A + B + C)
```

The trouble with this approach is that the sum of A, B, and C is not saved. If we want to use $A + B + C$ in a later computation, we would have to recompute it. So we only use expressions as output items when we are sure we won't need the value of that expression later in the program.

There are many occasions when we are interested not just in the results themselves, but in having the results printed in a certain way. For example, a payroll program that prints checks must print the payee's name on the proper line, the amount to be paid written in dollars and cents in the proper space, etc. As another example, a table of results should have numbers in nicely spaced columns, with identifying column headings. Controlling the appearance of the output is known as *formatting* the output.

Let us look at how Pascal allows the format of the output to be specified. One obvious way is by allowing us to specify what items go on each line of output. If, for example, we wanted to print values of SUM and DIFF on two separate lines, we would write

```
WRITELN(SUM);  
WRITELN(DIFF)
```

And if we wanted to double-space this output, we would write

```
WRITELN(SUM);  
WRITELN;  
WRITELN(DIFF)
```

Thus we insert a blank line between the two lines of printing.

Next we need to consider how the format of each line of output can be controlled. For example, the statement

```
WRITELN('X')
```

prints an X in the leftmost space of the line. Suppose we want to move it over. The statement

```
WRITELN(' X')
```

will put one blank space before the X, and, similarly, by putting 20 blanks before the X in the WRITELN statement, we can move the X over 20 spaces. This approach, however, is not only time-consuming but liable to error. (Counting blanks correctly is difficult.) There has to be an easier way! In Pascal, we can print X in the 21st column of a line by the statement

```
WRITELN('X':21)
```

The "21" specifies that we want to print 21 characters altogether. Since we have explicitly specified only one character, the other 20 are assumed to be blanks. Furthermore, the blanks are inserted *before* the specified character string.

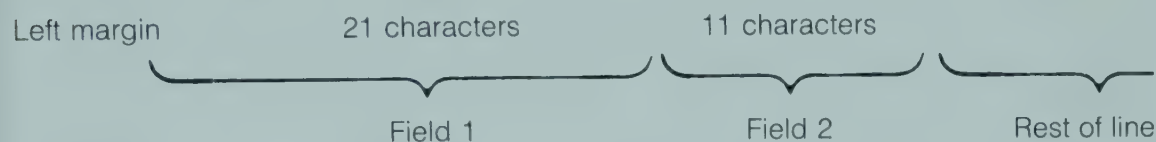
As another example, suppose we want to print 20 blanks, then "X", then eight blanks, then "XYZ". This can be done by the statement

```
WRITELN('X':21, 'XYZ':11)
```

This first prints a character string of length 21, consisting of 20 blanks followed by "X", then a character string of length 11, consisting of eight blanks followed by "XYZ".

In formatting a line of output, it is helpful to think of the line as

subdivided into *fields*, in each one of which we wish to print an item of output. The numbers after the colons in the `WRITELN` statement specify the widths of these fields. Thus the statement above specifies a field of width 21, followed by a field of width 11. Pictorially:



As we have seen, the specified character string will be printed in the right-most spaces in the field; that is, the character string is *right-justified* in the field. If we do not specify a wide enough field, the character string may be truncated. Thus on our computer the statement

```
WRITELN ('XYZ' : 2)
```

will print only the first two characters of the string, i.e., XY.

At this point you might suspect that the spacing of values of variables can be similarly controlled. Indeed, you would be right; however, the situation is more complicated. Not only must we consider how to control where on the line the numbers are printed but what the numbers themselves look like—e.g., how many digits after the decimal point are to be printed. Let us, then, look in detail at the question of formatting the printing of REAL numbers.

If you did one of the programming exercises at the end of the last chapter, you probably found that the REAL variables were printed out in a particularly ugly format. (This was the default format, i.e., the print format used by the system unless you specify otherwise.) The default format for a REAL variable is to print it in what is known as *scientific notation*. You should recall from high school algebra that in this notation numbers are expressed as a number between 1 and 10 times a power of 10. For example, 545.96 is written as 5.4596×10^2 , and 0.0091 is written as 9.1×10^{-3} . In computing jargon, this way of representing numbers is often referred to as *floating point*. This format allows representation of numbers over a very large range of sizes. There is a problem with this format. Since computer printout is all on one line, we need to have some way other than a superscript to denote the power of 10. The standard way is to preface the value of the power by E (which stands for exponent). Thus 9.1×10^{-3} might in fact be printed as 9.1E - 003. The specific default format (numbers of digits in fractional part and exponent) may vary somewhat from system to system. We assigned X the value 4200.50 and executed the statement

```
WRITELN('TOTAL TAX IS', X, ';')
```

to see what our system would do. The result was

```
TOTAL TAX IS 4.2005000000000E+003;
```


As we said, rather ugly! Notice the details of this format. For each number printed, there are 13 digits after the decimal point. Then follows an E, followed by a signed three-digit integer giving the value of the exponent. Although they are not so visible here (but see below), there are also two blank spaces preceding each number. One of these spaces is for the sign, in case the number is negative. The other serves to separate numbers, in case we want to print several in a row. In all, the *default fieldwidth* (total number of characters required to print a number in default format) on our system is 22.

The simplest way to discover and to display clearly what the default format is on any system is to write a short program such as the following:

EXAMPLE 4-1

```
PROGRAM RW(OUTPUT);
VAR V1,V2:REAL;
BEGIN
  V1 := 1.2;
  V2 := -1.2;
  WRITELN('123456789012345678901234567890');
  WRITELN(V1);
  WRITELN(V2)
END.
```

The first WRITELN statement here simply numbers the columns 1, 2, 3,... so that we can clearly see the formatting of the numbers subsequently printed. On our system the output of this program is

```
123456789012345678901234567890
  1.20000000000000E+000
 -1.20000000000000E+000
```

Run this program on your computer to determine its default output format for REAL numbers.

Exponential format is useful when you are working with very large or very small numbers or are doing scientific or engineering calculations. However, printing 14 digits (or whatever the default is for your system) will not usually make sense. In general, you want to look at how many digits there are in the data values you input, and only print out as many digits as will be meaningful in your result. To print a number in exponential format but with fewer than 14 digits, you can set a fieldwidth just as for character strings. Thus we find that, if X is a positive number,

```
WRITELN (X:10)
```

causes the value of X to be printed with one digit after the decimal point (plus two blanks and a digit before the point, the point itself, and five characters to

represent the exponent—for a total fieldwidth of 10). A fieldwidth of 10 is the smallest possible for this format. Your system may work somewhat differently; for example, it may omit one or both blanks or print only two digits for the exponent. Experiment until you understand how the print formatting works on your system. Use Example 4-1 as a model and vary it by including formatted `WRITELN` instructions.

Even though it is not too difficult to get used to reading numbers in exponential format, there are clearly many times when scientific notation is inappropriate. Such numerical information as taxes and costs are better displayed without exponents. For example, one might want to print a bill of sale in a format something like the following:

```
PRICE :      403.99
TAX :       20.25
TOTAL BILL : 424.24
```

To accommodate this requirement, Pascal includes a fairly simple technique to allow you to specifically print formats like this. Suppose you want to print a `REAL` variable `X`, with value 29.425, in just that format. The instruction you use is

```
WRITELN(X:7:3)
```

The first integer after the `X` (i.e., 7) is the *minimum fieldwidth*; this is the minimum number of characters to be printed, including decimal point, sign (if negative), and (on our system) at least one blank at the beginning. The second integer (3) is the *fraction length*. This specifies the number of digits to be printed after the decimal point. Notice that the specified fieldwidth is a *minimum*. If `X` were 1029.42, printing it would take nine characters. (There would still have to be three digits after the decimal point, if that's what we specify.) In many programming languages, specifying too small a field for printing a result causes drastic problems (e.g., you may get an error message instead of your result), but in Pascal, the fieldwidth is automatically increased to fit the number. (This may be at the expense of making your output look sloppy. You are well advised to determine ahead of time how big your results will be!)

For comparison purposes, we executed the statement

```
WRITELN('TOTAL TAX IS', X:9:2, ' ; ')
```

The output of this was

```
TOTAL TAX IS 4200.50;
```

Again, expand Example 4-1 to print some numbers in this format, so that you thoroughly understand what your system will do. Fill in the following table as a permanent record of how your system prints `REAL` numbers.

X = 1.2, Y = -1.2

Column	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
WRITELN(X)																										
WRITELN(Y)																										
WRITELN(X:10)																										
WRITELN(X:5)																										
WRITELN(X:12:2)																										
WRITELN(Y:15:5)																										

Before leaving this section, let us point out one more thing you should know about printed output. Suppose the variable X has the value 1.2. Then the statement

`WRITELN(X:12)`

will print

1.200E+000

as described earlier. But if you let X be printed in default format, i.e., by

`WRITELN(X)`

you *might* get

1.20000000000000E+000

On the other hand, you might get

1.19999999999999E+000

This effect is caused by the way numbers are stored in the computer memory. That is, it may not be possible to store 1.2 exactly, but only to store 1.19999999999999, which differs from 1.2 by only 10^{-13} . Explaining in detail how numbers are stored in memory is beyond the scope of this text. We will, however, say a bit more about this subject in Chapter 5. Meanwhile, keep in mind that the output of such strings of 9's does not signify an error. Just mentally round the number and don't worry about it! Notice that when you format your output so that the full fractional part is not printed (e.g., X:12 or X:6:2) rounding automatically takes place.

4-3 OUTPUT WITH WRITE

We will see examples later where the output to be printed on one line is more naturally generated character by character (or item by item) than prespeci-

fied by a single WRITELN statement. Pascal therefore includes a WRITE statement to allow you to print only part of a line at a time.

We can explain the difference between WRITELN and WRITE very easily. The LN (short for "line") after the WRITE simply tells the computer that *after* the items in parentheses are printed, the printer is to be advanced to the left margin of the next line. The LN has the effect of pushing the carriage return lever on a typewriter. On the other hand, WRITE just prints the items in the list and does not follow the items by a carriage return. Thus a subsequent WRITE will continue printing on the same line. You can sit down at a typewriter and simulate the effect of a sequence of WRITE and WRITELN instructions. Consider the following sequence:

```
WRITE('THE QUICK BROWN FOX JUMPED');  
WRITE(' OVER THE LAZY DOG. ');  
WRITELN;  
WRITELN('THE END.')
```

Given the first instruction, you start at the left margin and type:

```
THE QUICK BROWN FOX JUMPED  
                               ↑
```

The arrow here indicates the space where you are ready to type next. After the second instruction (notice that the character string to be typed begins with a blank, so that you will leave a space between JUMPED and OVER), you will have typed

```
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.  
                                              ↑
```

Again, the arrow indicates the current typing position. The third instruction (WRITELN) causes you to push the carriage return lever (without typing anything else), so then the situation is:

```
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.  
↑
```

The last instruction is also a WRITELN—so you type the given character string and then push the carriage return lever. Now you have

```
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.  
THE END.  
↑
```

If the program contains any further WRITE or WRITELN instructions, you will begin typing at the left margin on the third line, as the arrow indicates.

In most situations, it is adequate (and even simpler) just to use `WRITELN`. (As an exercise, convert the above set of four instructions to *two* `WRITELN` instructions.) But keep in mind that Pascal does give you a choice of whether to move on to the next line.

4-4 INPUT WITH READ AND READLN

When we want to read data into a computer, we must specify two things: the source of the data, i.e., the input device from which the data is to be read; and where the data is to be put, i.e., the names of the memory locations where we wish to store the data. If you are working at a terminal, the computer will expect the data to be input via the terminal. If you are submitting a program punched on a deck of cards, the computer will assume that the input data will also be punched on cards and that the cards containing data will be appended to your program deck. In either case there is a standard default input device, so generally you need not specify the input device.

Just as there are two instructions—`WRITE` and `WRITELN`—for output, there are two instructions—`READ` and `READLN`—for input. The difference between `READ` and `READLN` is quite analogous to the difference between `WRITE` and `WRITELN`. That is, the `LN` of `READLN` causes a skipping to the next “line” (or card) of input. However, as we shall explain, you don’t *need* to use `READLN` to move on from one line of input to the next. `READ`, therefore, is the most heavily used input instruction, and we shall begin with a discussion of `READ`.

A single data item can be input by an instruction such as the following:

```
READ (DIAMETER)
```

This statement will read a number from a card and store it in the memory location named `DIAMETER`. Thus, to compute the circumference of a circle, we might write the following program:

EXAMPLE 4-2

```
PROGRAM CIRCLE (INPUT,OUTPUT);
CONST PI = 3.14159;
VAR CIRCUM, DIAMETER : REAL;
BEGIN
    READ (DIAMETER);
    CIRCUM := PI * DIAMETER;
    WRITELN ('CIRCUMFERENCE IS', CIRCUM:10:4)
END.
```

Notice the heading of this program. We have followed the program name by

```
(INPUT,OUTPUT)
```

We needed to do this on our system because the program has both input and output; i.e., it contains both READ and WRITE statements. This program will read in a REAL number for the diameter, then compute and print the circumference. If we are using punched cards to input program and data, the data card might look, for example, like Figure 4-1.

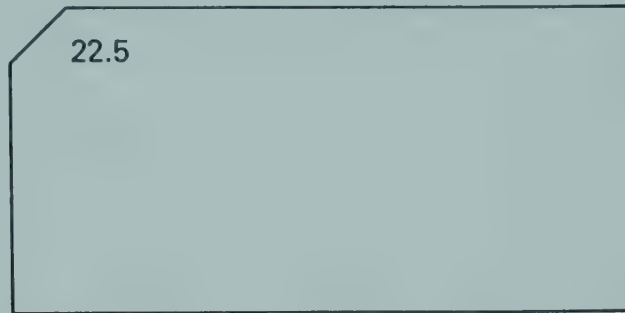


Figure 4-1

This data card must naturally follow the cards on which we have punched the program, since the program must be already in the computer before its READ statement can be executed. We have not specified in which columns of the card the number is typed, since the number can appear anywhere on the card.

If we are using a terminal to input program and data, the computer will stop when it comes to the READ statement and ask us to enter a number. Our system displays a question mark (?) on the terminal screen to indicate that it is waiting to read a data value. (Your system may well not do so.) We then type in the value

22 . 5

followed by a (carriage) *return* (to tell the computer that we have finished typing). This value is read, the computation is carried out, and the result (as specified by the WRITELN statement) is displayed on the screen.

Now suppose we have two numbers to read in—say we are computing the volume of a cone and need both the radius of the base and the height. We write

```
READ (RADIUS);  
READ (HEIGHT);
```

These instructions say to first read a value for radius, then for height. The two values can be put on the same card, as in Figure 4-2.

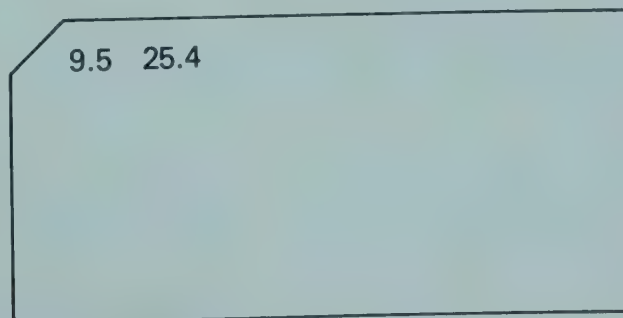


Figure 4-2

Notice that we must leave at least one space between the numbers, for the obvious reason. Similarly, working at a terminal you can type both numbers on a single input line, typing after the question mark:

9 . 5 25 . 4

followed by a *return*. Whether the input is on a punched card or typed at a terminal, what happens is that the first READ statement reads in a number, stopping at the blank space; then the computer will wait for further READ instructions to read any further numbers on the card or line. The two numbers can also be put on separate cards, as in Figure 4-3.

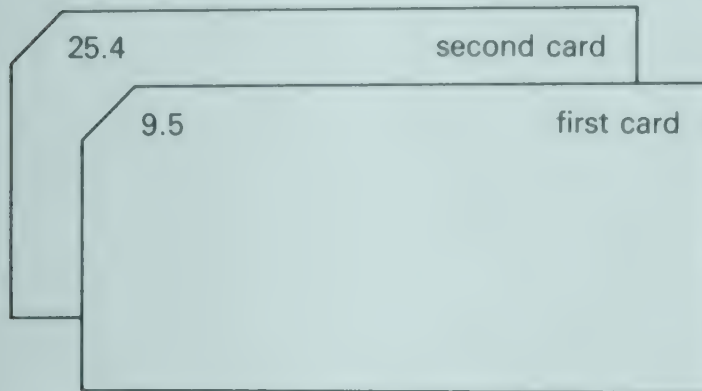


Figure 4-3

When the input is from cards, successive READ statements simply look for the next number, skipping over blanks, and proceeding to the next card when all numbers on the one card have been read.

When you are typing the input at a terminal, you can also type the numbers on two separate lines. What will happen is that after you type the first value

9 . 5

followed by *return*, the computer will display another question mark on the screen to indicate to you that it needs another number to read. Thus, after you have typed in these two numbers on the terminal, the following will have been displayed on the screen:

? 9 . 5
? 25 . 4

Just as for WRITE, you can include more than one item to be read in a single READ statement. Thus, the statement

READ (RADIUS, HEIGHT)

is *completely* equivalent to the two statements

READ (RADIUS);
READ (HEIGHT)

In particular, even if several items are specified in a single statement, the computer will automatically proceed from one card (or line) to the next, reading numbers until a value has been provided for each of the variables specified in the list. The fact that READ instructions automatically read data from successive cards (or lines) is why it usually suffices to use READ statements in your programs and not worry about READLN. However, let us look briefly at the effect of READLN.

Analogous to WRITELN, READLN reads a line (or card) of input. The effect of

READLN(RADIUS)

is to read a single value and then skip to the next card. In this case the height *must* be on the next card. If you type the two numbers on the same card, as in Figure 4-4

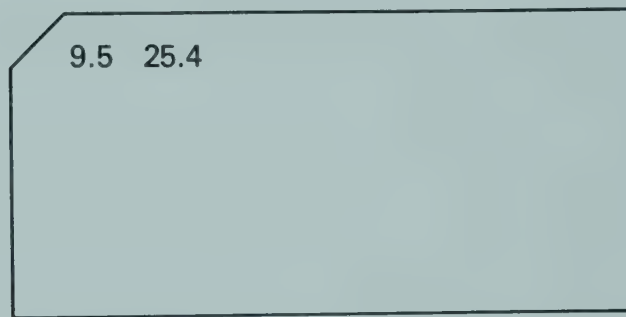


Figure 4-4

then the instruction READLN(RADIUS) will read the 9.5, store it in the location named RADIUS, and then skip to the next card, ignoring the 25.4. If you then have the instruction READLN(HEIGHT), this instruction will have no number to read. You will probably only want to use READLN if you are using punched card input and if there is something on the data card you want to skip over. READLN can be useful, for example, if you want to type extra information on the data cards by way of identification. (This can be helpful if you drop the cards and need to unscramble them.) You might want to type your data cards as shown in Figure 4-5

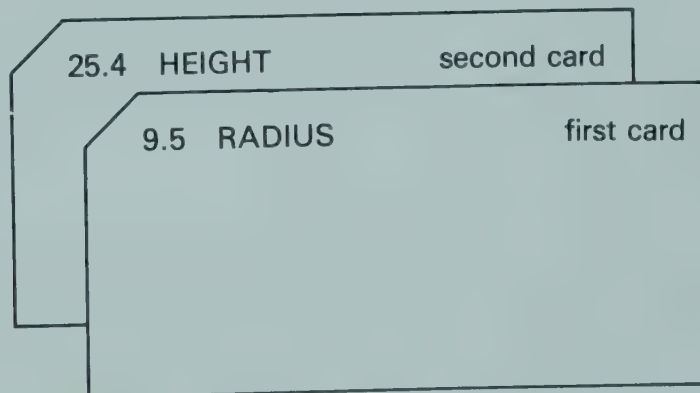


Figure 4-5

and input the values with the instructions

```
READLN(RADIUS);  
READLN(HEIGHT)
```

Let us look at one more example. Suppose the data cards are as in Figure 4-6.

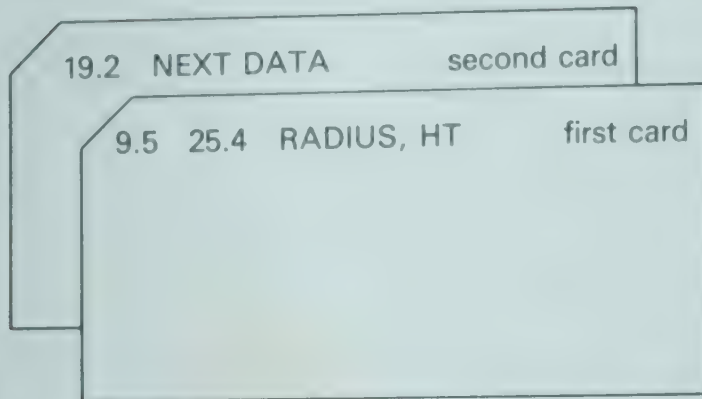


Figure 4-6

and suppose we want to skip over the alphabetic identifiers, while reading the three numbers into locations named RADIUS, HEIGHT, and NEXT. One way to do this is to write:

```
READ    (RADIUS);
READLN  (HEIGHT);
READLN  (NEXT)
```

The first READ statement reads 9.5, stores it into the memory location named RADIUS, and stops. The second statement reads 25.4, stores it into the memory location named HEIGHT, and skips to the beginning of the next line. The third statement reads 19.2, stores it in the memory location named NEXT, and sets the position of the next read (if there is any) to be at the beginning of the next card (if you have typed one). You could, of course, equally well use

```
READ (RADIUS);
READLN (HEIGHT);
READ (NEXT)
```

in the case that there are no further reads. But it is a dangerous and untidy practice to leave the reader in the middle of a card, ready to read junk if you should happen later to add an extra read instruction to your program.

Finally, note that another sequence of instructions you might use is

```
READLN (RADIUS, HEIGHT);
READLN (NEXT)
```

This sequence is the obvious analog of printing one line at a time by using successive WRITELN statements.

4-5 WRITING INTERACTIVE PROGRAMS

In the last section we explained how the READ and READLN statements are used to input data that is typed, either on cards or at the terminal, as

successive lines of data. We have also described how on our system a question mark is displayed on the terminal screen when a READ statement is being executed and we need to type a data item to be read. This question mark is known as a *prompt*, since it is prompting us to enter data. A program with which we interact (it prompts us for input, we respond, then it prompts us for more input or prints results, etc.) is known as *interactive*.

If you have been working at a terminal, you have already used interactive programs that are part of the computer system. For example, when you log onto the computer a program prompts you for your name or identification number and then probably requests your password. And in typing your program into the computer, you are prompted for successive program lines, etc. You have also probably noticed that the prompts are often messages and are generally more helpful than just a display of a single character like the question mark. You have already learned enough Pascal to write programs with helpful prompt messages. What you have to do is to use WRITELN to display an appropriate message on the terminal screen. For example, suppose you want to write an interactive program that computes the volume of a cone. You want to prompt the user of the program first to enter the radius, then the height. Instructions to do this will look something like the following:

```
WRITELN ( 'ENTER RADIUS' );  
( * PROMPT USER TO TYPE VALUE OF RADIUS * )  
READLN ( RADIUS );  
WRITELN ( 'ENTER HEIGHT' );  
( * PROMPT USER TO TYPE VALUE OF HEIGHT * )  
READLN ( HEIGHT )
```

Unfortunately, computer systems differ widely in how interaction with the terminal is handled and hence how interactive Pascal programs must be written. Thus the sequence of instructions written above, although it appears straightforward, is unlikely to work the way you would expect. Some systems require that an extra READLN instruction (without input list) be inserted after each prompt display with WRITELN. This is the case with our system; so our interactive program to compute the volume of a cone is the following.

EXAMPLE 4-3

```
PROGRAM CONE ( INPUT / , OUTPUT );  
CONST PI = 3.14159;  
VAR RADIUS, HEIGHT, VOLUME : REAL;  
BEGIN  
  WRITELN ( 'TO COMPUTE VOLUME OF CONE' );  
  WRITELN ( 'ENTER RADIUS' );  
  READLN;  
  READ ( RADIUS );
```

```
        WRITELN ( 'ENTER HEIGHT' );  
        READLN;  
        READ (HEIGHT);  
        VOLUME := HEIGHT*PI*RADIUS*RADIUS/3.0;  
        WRITELN ( 'THE VOLUME IS', VOLUME:8:3 )  
    END.
```

When we ran this program, what was displayed on the terminal was

```
    TO COMPUTE VOLUME OF CONE  
    ENTER RADIUS  
    ? 3.5  
    ENTER HEIGHT  
    ? 2.5  
    THE VOLUME IS 32.070
```

Here we have underlined the two numbers that we entered.

As an extra peculiarity on our system, notice that we put a slash after the word `INPUT` in the program header to indicate that the program is interactive. Your instructor will have to supply you with the information you need to write interactive programs on your system and also with such details as whether your system displays a question mark when it expects input. Because of the wide differences between systems, most of the examples in this text will not be written as interactive programs. But interactive programs are used extensively in the real world. (Notice that the user is told what to do and so doesn't need to understand the program at all.) Hence you want to be sure to get some experience in writing such programs.

4-6 NOTES ON PROGRAMMING STYLE AND PITFALLS

Let us look more closely at Example 4-2—the program that computes the circumference of a circle. It looks all right; indeed it will carry out the computation we are interested in with no problem. Nevertheless, there are several ways in which it falls short of being a *good* program.

First, notice the output statement. Judging from the format, it looks as if we are expecting to compute circumferences of less than 10,000. (We have only allowed room for four digits before the decimal point.) This implies a limit on the diameter. Although in this case nothing too dreadful will happen if we run over the limit (recall that the fieldwidth will be automatically lengthened if necessary), it is good practice to indicate to the reader what **limits on the input variables we have in mind**.

One way to do this is by a comment. Recall that comments serve to annotate or *document* a program, and that it is standard programming practice to introduce a program with a comment (possibly quite lengthy).

which describes what the program does. This comment should also include other information useful to potential users, such as data limitations. Thus our example might begin:

```
( * PROGRAM TO COMPUTE CIRCUMFERENCE      * )
( * OF A CIRCLE OF GIVEN DIAMETER.        * )
( * DIAMETER SHOULD BE LESS THAN 1000.    * )
( * UNIT OF LENGTH ARBITRARY;              * )
( * RESULT WILL BE IN SAME UNITS.          * )
```

Notice that we have bracketed each line of this multiline comment separately. This was not necessary; we could have omitted all the brackets between the first and last. However, most programmers find that it makes programs easier to read when the comments are clearly set off from the program as illustrated here. As we noted in Chapter 3, in a long program other comments can profitably be inserted to help the reader. On the other hand, too many comments explaining things that are obvious are only distracting. Providing just the right amount of documentation is a part of the programming job that is worth putting some time and thought into.

The second thing wrong with our program is that, although it clearly identifies the result as the circumference of a circle, it doesn't tell us what the diameter is. We can fix this by changing the output statement to

```
WRITELN ( 'CIRCUMFERENCE OF CIRCLE WITH DIAMETER' ,
          DIAMETER:9:4 , ' IS' , CIRCUM:10:4 )
```

Now if we read the output next week or next month, we will still have a clearly identified, meaningful result. We could, of course, try to remember to always scribble down the input data values on the printed output, but this is one of the jobs it is best to let the computer do for us. It not only serves to identify the output but also convincingly demonstrates that the result was computed using the correct data. (We could, after all, have made a mistake in the data card!) There may be cases where it seems impractical to write out all the input data—e.g., if you are computing an average of 10,000 numbers. But as a general rule, *include your input data with your printed results for identification and checking purposes.*

Now that we have mentioned the possibility of bad input data, this is a good time to pursue that point further. Most of the dramatic cases of “computer errors” that you read about in the newspapers are not really computer errors—or even program errors—they are data input errors. For example, a clerk may type a card indicating that a certain worker worked 400 hours (instead of 40) one week. And the computer unwittingly multiplies by the hourly wage (say, \$10) and generates a check for \$4000. Where should the blame fall—on the computer, the clerk, or the programmer? In fact, it should fall squarely on the programmer! *A good program should check for bad or unreasonable data.*

In the case of a program to print paychecks, there could be a maximum possible number of hours worked. Or there could be an upper limit on the size of the check that can be generated. If a specified limit is exceeded, an error message could be printed. In the case of our program to calculate the circumference of a circle, we want the diameter to be a *positive number less than 1000*. The Pascal language has statements, which we will tell you about later, that allow us to easily write such a check into our program. Another standard check is to make sure there is data there at all. We may have forgotten to include the data card, or we may have typed too few numbers on the card. Attempting to READ nonexistent data will cause a program to abort. The computer may or may not provide us with a clear indication of the source of the error. It is much better practice to include in our program the action to be taken in case there is not enough data. We have not yet come to the instructions that you need to write such checks into your program. But keep in mind as you write simple programs that such checks are really needed. Always decide what constitutes valid input data for your program, and, until you learn how to write validity checks into your program, include at least an appropriate comment.

4-7 EXERCISES

4-1. Convert the following values to a more easily readable form:

- (a) 1.2350000000000E+000
- (b) 6.4000000000000E+002
- (c) -5.7130000000000E+001
- (d) 4.9700000000000E-002

4-2. Convert the following values to "E form"—i.e., the Pascal default print form for REAL values. (Use the correct format for *your* system.)

- (a) 27.3
- (b) -365.0
- (c) -.004
- (d) 1,000,000.0
- (e) .00006

4-3. What would be printed by the following sequence of statements (or program segment)?

```
VALUE1 := 27.3;
VALUE2 := -6.5;
WRITELN('VALUE 1 IS ', VALUE1);
WRITELN('VALUE 2 IS ', VALUE2 : 16);
SUM := VALUE1 + VALUE2;
WRITELN('AND THEIR SUM IS ', SUM : 6:2)
```

4-4. Given the data cards or lines below, what would be printed by the following program segments?

```

(a) READ(FIRST);
    READLN(SECOND);
    READLN(THIRD);
    READ(LAST);
    Writeln('THEY ARE:');
    WRITE(FIRST : 6:2, SECOND : 6:2);
    Writeln(THIRD : 6:2, LAST : 6:2)

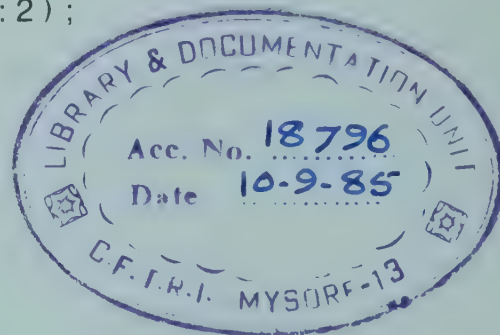
(b) READLN(FIRST);
    READ(FIRST, SECOND);
    READLN(SECOND);
    READ(THIRD);
    READLN(LAST);
    WRITE('THEY ARE ');
    Writeln(FIRST : 6:2, SECOND : 6:2, THIRD : 6:2);
    Writeln(LAST : 6:2)

(c) READLN(FIRST, SECOND);
    READLN(THIRD, SECOND, LAST);
    READ(THIRD, LAST);
    READLN(THIRD);
    READLN(LAST);
    Writeln(FIRST : 6:2);
    Writeln(SECOND : 6:2, THIRD : 6:2);
    Writeln(LAST : 6:2)

```

Data

13.0	14.5	15.6	16.3
9.24	10.3	11.4	12.3
5.32	6.79	7.82	8.13
1.64	2.39	3.12	4.56
4.64	4.10	5.10	5.83



- 4-5.** Draw a syntax diagram for the READ statement as described in this chapter.
- 4-6.** Draw a syntax diagram for the Writeln statement as described in this chapter. (Since any allowable character may be printed out, use a rectangular box labeled "character" in your syntax diagram.)
- 4-7.** We wish to be able to evaluate expressions of the form $ax + by + c$. Write a program that will read the values a , b , c , x , and y from data cards (or terminal input lines) typed as follows:
- | | | | |
|-----|-----|-----|---------------------------------|
| a | b | c | COEFFICIENTS (1st card or line) |
| x | y | | VARIABLES (2d card or line) |
- The program should then print these values, compute the value of the expression $ax + by + c$, and print this value.
- 4-8.** Write a program that reads three values: x , y , and z . These values should be between 0.0 and 99.9. Print each value four times using four different formats. Also print the

values of $x + y + z$ and $x * y * z$ using both floating point and fixed point formats. Be sure to label every value that is printed, so that the output can serve as an example of the various formats available.

- 4-9.** Write an interactive program that prompts a user for his or her social security number (SSN), height, and weight and displays weight per inch of height, identified by SSN. Make sure your prompt messages tell the user in what units you want height and weight specified.
- 4-10.** Write an interactive program to be used by a clerk in the tax collector's office to compute tax bills.

Input: Taxpayer identification number
Assessed valuation
Tax rate

Output: A nicely formatted bill, containing all the input data plus the tax owed

SUMMARY OF CHAPTER 4

The instructions used for input and output are

WRITELN (list of items)
WRITE (list of items)
READLN (list of items)
READ (list of items)

Items to be written may include constants, variables, and expressions. The "list of items" in a READ or READLN instruction includes only variable names: values are then read and assigned to these variables in the specified order.

WRITELN writes a line. Thus any subsequent WRITE statement will start writing on a new line. On the other hand, WRITE only writes part of a line. A subsequent WRITE statement will continue on the same line. Similarly, READLN reads a line (or a card). No matter whether all the data values in the line are read, a subsequent READ statement will read from the next line. (Thus if there are unread data values in the first line, they are ignored and cannot be retrieved later on in the program.) READ will not skip over any values. Both READ and READLN read values from as many lines as are necessary to provide values for all the variables in the input list.

For some systems the words INPUT and OUTPUT must appear in parentheses following the program name in the heading of a program that contains READ and WRITE statements.

5

INTEGERS AND INTEGER VARIABLES

5-1 INTRODUCTION

You may have observed that throughout our discussion of arithmetic in Chapter 3 we were always very careful to write every number with a decimal point—even numbers like 2.0 or 100.0. You may have wondered why we did not simply write “2” or “100.” There was a reason—namely, we were discussing arithmetic expressions involving REAL variables and REAL constants. In Pascal, a REAL number must be written with a decimal point. For reasons that will be given in the next section, most computers handle two critically different *types* of numbers—REAL and INTEGER. These two types of numbers are stored and manipulated inside the computer in different ways. Thus numerical quantities such as two or one hundred can be handled inside the computer either as REAL numbers or as INTEGER numbers. When we write a number with a decimal point—like 2.0—we are telling the computer that the numerical quantity is to be treated as REAL. When we omit the decimal point and simply write “2” we are saying that the numerical quantity is to be handled as an INTEGER.

5-2 REPRESENTING NUMBERS IN A COMPUTER

Recall that each memory location in a computer holds a single number. Although there might be some slight variations in some computers, a rather accurate description of the situation is that a location can hold only a fixed

number of digits. For example, let us assume that a location can hold five digits. Each location can then hold an integer between 00000 and 99999, inclusive. For example, 81234 would be stored as

8	1	2	3	4
---	---	---	---	---

where we have indicated each of the five digits storable in a memory location by a separate box. But suppose that we also need to be able to handle fractions and numbers larger than 100,000. One easy way to handle such numbers is to represent them as a fraction times a power of 10, i.e., in scientific (or exponential) notation, as discussed in Chapter 4. If we have only five digits to work with, we might use the first four digits for the fraction and the last digit for the exponent. Thus, for example, 1.025 (0.1025×10^1) would be stored as

1	0	2	5	1
---	---	---	---	---

Fraction Exponent

and 102000 (or 0.102×10^6) would be stored as

1	0	2	0	6
---	---	---	---	---

Using this representation, the smallest nonzero number that can be handled is 0.0001×10^0 , which is stored as

0	0	0	1	0
---	---	---	---	---

and the largest is 0.9999×10^9 (or 999,900,000), stored as

9	9	9	9	9
---	---	---	---	---

This looks like quite an improvement, since the range of numbers that we can handle has been extended enormously.*

*In real computer memory, minus signs can be attached to both fraction and exponent. A minus sign attached to the fraction allows us to represent negative quantities like -0.502×10^3 . A minus sign attached to the exponent allows us to represent quantities, like 0.002×10^{-9} , that are smaller in magnitude. However, we shall not complicate our description of number representation with these additional details.

But notice the price we have paid for this extension. We have sacrificed some of the precision with which we can represent numerical quantities. For example, we could store the number 73516 exactly as a five-digit integer:

7	3	5	1	6
---	---	---	---	---

But since one of these five places is needed to hold the exponent, we cannot exactly store 73516 in exponential representation. What happens is that we can write 73510 as 0.7351×10^5 and store it as 73515. We can store 73520 as 73525. However, all the numbers in between, namely, 73511, 73512, 73513, ..., 73519 cannot be represented precisely but must be approximated by either 73510 or 73520 (whichever is closer) and stored accordingly.* Since 73511, 73512, ..., 73519 are just integers that could be stored exactly if one of the digits weren't used for an exponent, it would be unfortunate if we were forced to lose this accuracy by having to store all numbers in the exponential format. Most computers therefore allow both types of representation: exponential (referred to as REAL numbers in Pascal) and integer (referred to as INTEGER numbers in Pascal). This flexibility allows us to take advantage of the extra accuracy possible when we are working only with integers. Also, INTEGER numbers are handled more efficiently by the computer. Like you, the computer has to work a bit harder to do arithmetic on numbers in exponential format.

Now there is another problem. When we peer into a memory location in the computer, we see only a sequence of digits. For example, a location may contain

1	2	3	4	2
---	---	---	---	---

Is the number stored here 0.1234×10^2 or 12342? The only way that we—and also the computer—can tell the difference is to have this memory location identified not only with its name but with its data *type*, either REAL (stored in the exponential format) or INTEGER (stored in the integer format).

For each variable name used in our program, we must declare whether the memory location identified by that name is to hold the value of a REAL variable or of an INTEGER variable. You should recall from Chapter 3 that program blocks in general begin with such declarations. Thus we might write

```
VAR BUDGET, POPULATION: REAL;
    NUMBERDAYS: INTEGER;
```

This is the first example you have seen of a declaration in which two different types of variables are declared. Look back at the syntax diagram in Section 3-5 to verify that this example correctly illustrates how to do it.

*It is this sort of inability to store REAL numbers exactly that can cause the printing anomalies noted at the end of Section 4-2.

Similarly, when we use constants in our program, there must be a way to distinguish whether they are to be stored as REAL or INTEGER numbers. This is done simply by always including a decimal point in REAL numbers. Thus 100.0 is stored as

1	0	0	0	3
---	---	---	---	---

while 100 is stored as

0	0	1	0	0
---	---	---	---	---

This convention to distinguish between the two types is used when we declare constants, use constants in expressions, or type them as input. For example,

```
CONST X = 10;  
      Y = 24.0;
```

defines X to be an INTEGER constant with value 10 and defines Y to be a REAL constant with value 24.0. Then if we write

```
5 * X
```

we are multiplying the INTEGER X by the INTEGER 5 to give an INTEGER 50.

5-3 TYPE ERRORS AND TYPE CONVERSIONS

If we attempt to use the wrong type in the wrong place, one of two things can happen. Either the computer will tell us that we have made a *type error* or it will perform a *type conversion* (i.e., a conversion of a numerical quantity from one representation to another), which may or may not yield the number we want. For example, suppose we declare

```
VAR NUMBER: INTEGER;
```

and later we have a

```
READ (NUMBER)
```

and the number to be read is typed as

```
25.32
```

Clearly, we are trying to read a REAL number into an INTEGER location. The computer may stop our job and tell us that we have made an error. Or it may casually throw away the fractional part (.32) and simply store 25 without notifying us. It is extremely important to make the input data format agree with the variable type. Additional complexities can arise when we use INTEGERS—or a mixture of INTEGERS and REALs—in expressions. We will discuss these problems in detail in Section 5-4. First, let us look closely at how type conversions are carried out inside the computer.

Consider, for example, the assignment statement

REALNUM := 65

where REALNUM has been declared as a REAL variable. Here we are trying to store an INTEGER constant into a location designated as REAL. In a memory location that can hold five digits, 65 is represented as

0	0	0	6	5
---	---	---	---	---

Before this number can be stored in REALNUM, it must be converted to its REAL representation

6	5	0	0	2
---	---	---	---	---

Just as you must go through a bit of mental figuring to convert

0	0	0	6	5
---	---	---	---	---

to

6	5	0	0	2
---	---	---	---	---

the computer also must do some work to carry out the conversion. That's why we are careful to make the types the same and write

REALNUM := 65.0

even though it is perfectly legal to assign an INTEGER constant to a REAL variable.

Conversion of a number from INTEGER to REAL can, as we noted before, also cost us some accuracy. Thus, if we write

REALNUM := 64999

the constant

6	4	9	9
---	---	---	---

 must be converted to

6	5	0	0	5
---	---	---	---	---

 or

6	4	9	9	5
---	---	---	---	---

to be stored as REALNUM. Which of these REAL representations the computer uses will depend upon whether it rounds the integer before dropping the last digit.

Sometimes one might wish to convert numbers from REAL to INTEGER as well as from INTEGER to REAL. Clearly, a REAL number can be converted into an INTEGER either by truncation—throwing away the fractional part—or by rounding the REAL value to the nearest integer. It is important, however, that the programmer understand exactly what has happened and that the conversion be exactly what he or she wanted to happen. In most Pascal implementations, therefore, REAL-to-INTEGER conversions that are not explicitly specified [by using TRUNC (truncate) or ROUND, as we will explain later] are forbidden. Thus, if you declare INTNUM an INTEGER variable and write

```
INTNUM := 65.2
```

you will get an error message, and your program will not run. Even if the REAL number has a zero fractional part, as in

```
INTNUM := 65.0
```

your program will not run.

This disallowance by Pascal of automatic REAL-to-INTEGER conversions can cause some subtle difficulties. As we will explain in the next section, certain expressions that to the uninitiated look as if they will evaluate to INTEGER values may in fact turn out to be REAL—so they cannot then be assigned to INTEGER variables!

As a general rule, make sure you understand and are always aware of the data types you are using. Keep allowable conversions (INTEGER-to-REAL) to a minimum because they are costly. And remember that REAL-to-INTEGER conversions are usually forbidden in Pascal.

5-4 ARITHMETIC EXPRESSIONS USING INTEGER VALUES

The symbols `+`, `-`, `*`, `/` are used to construct arithmetic expressions involving INTEGER variables and constants in the same way that they are used to construct REAL expressions. The same rules of precedence apply. But opera-

tions with INTEGERS pose some problems that don't arise for REALs. Let us remind the reader that adding one REAL value to another, subtracting one REAL value from another, multiplying one REAL value by another, and dividing one REAL value by another will always yield a result that is also a REAL value. Similarly, adding one INTEGER value to another, subtracting one INTEGER value from another, and multiplying one INTEGER value by another yield results that are INTEGER values. However, dividing one INTEGER value by another will yield a result that is a REAL value. This is desirable since, for example, 7 divided by 2 is 3.5, which must be stored as a REAL number.

Study carefully the following simple examples of arithmetic with INTEGER values.

Expression	Result	
$3 + 4$	INTEGER	7
$3 * 4$	INTEGER	12
$3/4$	REAL	0.75
$3 - 4 * 2$	INTEGER	-5
$4/2$	REAL	2.0

An obvious next question is: What happens when an expression such as $3 - 2.5$ is evaluated? Since 2.5 is a REAL number, the subtraction operator has one operand that is an INTEGER and one that is a REAL. In Pascal, the rule to cover this situation is quite simple: If any of the operations $+$, $-$, $*$, or $/$ is to be carried out with one operand a REAL value and the other an INTEGER value, the INTEGER is first converted to a REAL, after which the operation is carried out. The result will necessarily be a REAL value.

Thus, to evaluate $3 - 2.5$, the INTEGER 3 is first converted to a REAL 3.0. The subtraction is then carried out to yield a REAL 0.5. Similarly, to evaluate $3 - 3/2$ the division is carried out first to yield a REAL 1.5. The INTEGER 3 is then converted to REAL 3.0, and the subtraction is carried out. The final result is the REAL value 1.5.

As another example, consider the following arithmetic expression:

$$5 * 8/2 * 3 + 9$$

Evaluating the first term starting from the left, we multiply $5 * 8$, giving the INTEGER 40. The INTEGER 40 is then divided by the INTEGER 2 to give a REAL 20.0. This REAL is multiplied by 3 (converted to 3.0) to give REAL 60.0. The INTEGER 9 is then converted to a REAL and added to give a REAL total of 69.0. Notice that here is an example of an expression that only involves INTEGER constants and that evaluates to a result that has zero fractional part. But because the expression contains a division, the result is in fact a REAL value. Thus, if we write an assignment

```
INTNUM := 5 * 8/2 * 3 + 9
```

(where INTNUM is declared an INTEGER) our Pascal compiler will inform us that we are attempting an illegal type conversion, and our program will not run. Another point to notice about this expression is that, because of the type conversions that become necessary during its evaluation, we would have done much better to have written it in terms of REAL constants to begin with:

$$5.0 * 8.0 / 2.0 * 3.0 + 9.0$$

In Pascal, there are two additional operations, DIV and MOD, involving the division of INTEGER values. If I and J are INTEGER variables or constants, I DIV J will give the integral part of the quotient of I divided by J, and I MOD J will give the remainder of I after division by J. (It is illegal for one or both of the operands of these operators to be REAL.) The results of both of these operations are INTEGER values. For example, 47 DIV 5 is 9, and 47 MOD 5 is 2. These results follow from the fact that 5 goes into 47 nine times, leaving 2 as remainder. In carrying out a sequence of operations, the precedence of DIV and MOD is the same as that of * and /. Consider the following example involving these new operators.

$$84 \text{ DIV } 5 \text{ MOD } 6 * 3.3$$

Again, we carry out the operations left to right. 84 DIV 5 gives the result 16. 16 MOD 6 equals 4. The INTEGER 4 is converted to REAL 4.0 and multiplied by 3.3 to yield 13.2 as the final result. Notice that

$$4.5 * 7 \text{ DIV } 2$$

is illegal, because $4.5 * 7$ yields the REAL constant 31.5, which is illegal as an operand for DIV. On the other hand, we may parenthesize to form the legal expression

$$4.5 * (7 \text{ DIV } 2)$$

which evaluates to the REAL value 13.5. Table 5-1 summarizes the rules for arithmetic operations.

Now suppose we want to evaluate the expression

$$321.5 * 2 \text{ MOD } 8$$

This is illegal, of course. However, $321.5 * 2$ is 643.0, which, being an integer in the usual sense, is certainly meaningful to use as an operand for DIV and MOD. It is only the technicality that it is being represented in the computer as a REAL that makes 643.0 an illegal operand. What we need is some way to transform numbers from REAL to INTEGER representations. Pascal has two ways to accomplish this. First, you can use TRUNC(X) to get the integral part of X. (X is said to be *truncated*.) Thus TRUNC(3.7) is 3 and TRUNC(-3.7) is -3. Second, you can use ROUND(X) to get X rounded to the

Table 5-1
ARITHMETIC OPERATIONS

Operator	Type of operands	Type of result
+ (Addition)	Both INTEGER	INTEGER
	One or both REAL	REAL
- (Subtraction)	Both INTEGER	INTEGER
	One or both REAL	REAL
* (Multiplication)	Both INTEGER	INTEGER
	One or both REAL	REAL
/ (Division)	INTEGER or REAL	REAL
DIV (Division with truncation)	INTEGER	INTEGER
MOD (Modulo: remainder after division)	INTEGER	INTEGER

nearest integer. Thus `ROUND(3.7)` is 4 and `ROUND(-3.7)` is -4. In writing expressions, we need to be careful which one we use. Consider the expression

$$321.5 * 2 \text{ MOD } 8$$

If we want to assign this value to an INTEGER variable, `INTNUM`, do we write

```
INTNUM := TRUNC ( 321.5 * 2 ) MOD 8
```

or

```
INTNUM := ROUND ( 321.5 * 2 ) MOD 8
```

At first glance, it looks as if there should be no difference, since the expression in parentheses evaluates to 623.0. However, as we pointed out earlier, numbers are not always represented exactly in the computer. The number 623.0 may well look to the computer like 622.999, which would truncate to 622. Thus we must use `ROUND` to get the answer we expect. Remember to *think* before using `TRUNC` or `ROUND` to make sure you get the integer you want!

5-5 WRITING INTEGER VALUES

Recall that when REAL values are printed, the default fieldwidth (on our system) is 22 characters, and REAL values are always printed with a space before the number. In contrast to this, the default fieldwidth for INTEGER values is 10 (on our system), and no leading spaces need be printed. For

example, suppose we have an integer *I* with value 1234567. The statement

```
WRITELN(I)
```

will print three blanks followed by the seven digits of the number. Just as for REAL numbers, we can specify the minimum fieldwidth. Thus

```
WRITELN(I:7)
```

will print just the seven digits of *I*, while WRITELN(I:6) will also print the seven digits of *I*, since six characters are not sufficient. (Remember that Pascal increases the specified fieldwidth, if necessary, to fit the number to be printed.) If we want more spaces, we can write, for example,

```
WRITELN(I:15)
```

which will print eight spaces before the seven digits of the number, for a total fieldwidth of 15.

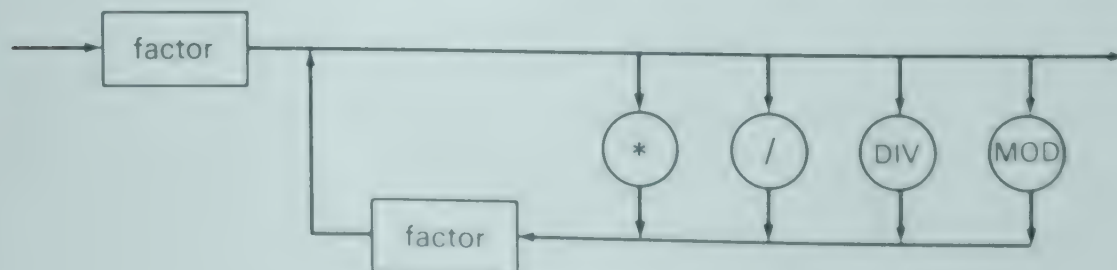
We emphasize that this is a description of how *our* system writes integers. Yours may be slightly different. In particular the default fieldwidth may be some number other than 10. Try the examples above on your system and see what happens. Use the program of Example 4-1 as a model, so you can clearly see the blanks. Fill in the table below so that you have a permanent record of the answers.

Column	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
WRITELN(I)																
WRITELN(I:7)																
WRITELN(I:5)																
WRITELN(I:15)																

5-6 SYNTAX DIAGRAMS

We have introduced two new arithmetic operations—DIV and MOD—in this chapter. We can easily expand the syntax diagram for *term* to include them.

Term



TRUNC and ROUND are actually *functions* that convert one number into another. We will look later at functions and the role they play in Pascal.

5-7 EXERCISES

- 5-1.** Assuming that numbers are represented in a computer in exactly the manner described in Section 5-2, how are the following values stored? (There may be more than one answer.)

3.14 86,000,000 14 912

What values could the following represent?

1	4	7	1	6
---	---	---	---	---

0	0	0	0	1
---	---	---	---	---

0	0	2	0	0
---	---	---	---	---

- 5-2.** Write a declaration for variables to hold the following values. (Remember to choose meaningful variable names.)
- (a) The number of students in the class
 - (b) The current year
 - (c) A random number between 0 and 1
 - (d) The day of the month
 - (e) Your grade point average
- 5-3.** Evaluate the following expressions. Is the value of each an INTEGER or a REAL?
- (a) $7 + 9 * 1.4$
 - (b) $3 - 6/2$
 - (c) $4 - 7 \text{ MOD } 3$
 - (d) $4 - 8 \text{ DIV } 2$
 - (e) $(7 \text{ MOD } 2)/(8 \text{ DIV } 6) + 4 * 7 \text{ DIV } 2$
 - (f) $4 * (7 \text{ MOD } 4) - 2$
 - (g) $\text{TRUNC}(1.6 * 9.2 + 3.1)$
 - (h) $\text{ROUND}(1.6 * 9.2 + 3.1)$
- 5-4.** Without using MOD or DIV, write assignment statements which produce the same results as $A := B \text{ MOD } C$ and $D := E \text{ DIV } F$.
- 5-5.** Write a Pascal program that first reads an INTEGER value which represents a number of days and then computes and prints the same number of days expressed as months, weeks, and days. (Assume each month has 30 days.) Your output might look like this:
- 83 DAYS IS 2 MONTHS, 3 WEEKS, AND 2 DAYS
- 5-6.** Temperatures reported on the TV news are always rounded to the nearest degree and are usually quoted in both Fahrenheit and Celsius. Write a Pascal program to do the following:
- (a) Read a Fahrenheit temperature given to the nearest tenth of a degree.
 - (b) Print one line showing °F as input (REAL); °F rounded to the nearest integer (INTEGER); and °F as input converted to °C and then rounded to the nearest integer (INTEGER).

EXAMPLE:

Input: 85.4

Computation: 85.4F rounds to 85F. Using the relation $\text{cels} = 5/9 (\text{fahr} - 32)$, 85.4F converts to 29.667C, which rounds to 30C.

Output: 85.4F rounds to 85F, converts to 30C (rounded).

- 5-7.** Write a Pascal program which converts a three-digit integer with digits XYZ into an integer with digits ZYX. (For example, 317 becomes 713.)
- 5-8.** In some systems, TRUNC does not work the way it should; for example, you may find that on your system $\text{TRUNC}(-3.7) = -4$. Write an interactive program that reads a REAL number X and displays TRUNC(X). Use the program to see what your system does with various values of X.
- 5-9.** Write a program that reads a REAL number giving a distance in miles and outputs the distance in kilometers, meters, yards, and inches. Be sure to identify the results.

SUMMARY OF CHAPTER 5

In Pascal there are two types of numerical values and, consequently, two types of variables, REAL and INTEGER. Some arithmetic expressions and assignment statements involve only REAL variables and constants. Others involve only INTEGER variables and constants. (But the division of two INTEGER values yields a REAL result and may lead to an expression that includes both REAL and INTEGER values.) When different types are involved in an arithmetic expression or in an assignment statement, either a type conversion will take place or a type-error alarm will be sounded. Note that operations TRUNC or ROUND can be used to convert REAL values to INTEGER values.

For integer arithmetic, besides the addition, subtraction, multiplication, and division operations, Pascal includes the DIV and MOD operations. $X \text{ DIV } Y$ is equal to the integral part of the quotient of X divided by Y. $X \text{ MOD } Y$ is equal to the remainder obtained when X is divided by Y to yield an integral quotient.



LOOPS AND THE FOR STATEMENT

6-1 INTRODUCTION

At this point, you have already gained some experience in writing programs and running them on a computer. You may have also noticed that a computer can execute a simple Pascal program consisting of several statements in a matter of a few seconds. An obvious question is: How much time must a programmer put in to write programs that will keep a computer busy for hours? Indeed, one might be led to think that only programs with hundreds of thousands of statements will consume any significant amount of computer time. But that's not so. In many (as a matter of fact, in all) nontrivial programs, simple tasks, corresponding to a small number of statements, are repeated many times. It is conceivable that it might take a few hours for a computer to repeat a short program of 20 statements 100,000 times. For example, a few Pascal statements, corresponding to a few seconds of computer time, will be sufficient to read in an employee's name, hourly wage, and number of hours worked in a week and to print the weekly paycheck for that employee. To repeat this task so that the paychecks of 100,000 employees are printed will certainly consume hours of computer time.

Recall how a grade school teacher keeps a restless student busy by giving the student the simple instruction, "Write 'I will be quiet in class' 500 times." Similarly, we might wish to instruct the computer to write the line `I WILL BE HONEST` 1000 times, or more realistically, to print grade reports for 35,000 students or to compute and print paychecks for 100,000 employees. The repetitive execution of a task by a computer is referred to as a *loop*. Any programming language must have ways to instruct the computer to carry out tasks repeatedly; that is, to control loops. This chapter explains one of the several ways available in Pascal.

6-2 THE FOR STATEMENT

A simple way to tell a computer (or anybody) to carry out a certain task repeatedly is to say something like, "Repeat the following 200 times." Thus, your physical education instructor might tell you to run 10 laps or do 20 push-ups.

Given such an instruction, you are undoubtedly very careful to keep count of the laps (or push-ups) as you do them, lest you inadvertently do too few—or even worse, too many! Similarly, when a computer is instructed to repeat a task a number of times, it must also have a means of keeping track of how many times the task has been repeated. A common way is to use an integer variable as a *counter* to count the number of repetitions. This variable is usually referred to as the *control variable* or the *index variable* or sometimes, for short, the *index*. Ordinarily, the value of the index variable is initially set to 1 and is then increased by 1 each time the task is completed. To instruct the computer to carry out a task repeatedly, Pascal contains a statement which explicitly specifies an index variable to be used to keep count as the task is repeated.

Consider the following example, which prints the line I WILL BE HONEST 1000 times.

```
FOR J:= 1 TO 1000 DO  
  WRITELN(' I WILL BE HONEST ')
```

In this example, the integer variable J is used as the index. The value of J is set initially to 1. (Thus 1 is called the *initial value*.) The task to be repeated is printing the line I WILL BE HONEST. Each time the line is printed, the value of J will be increased by 1. When the value of J exceeds 1000 (that is, when J equals 1001), the task will not be repeated any more. (Thus 1000 is called the *final value*.) We display in the following the sequence of steps to be carried out:

```
J is set to 1.  
Write I WILL BE HONEST.  
J is increased to 2.  
Write I WILL BE HONEST.  
J is increased to 3.  
Write I WILL BE HONEST.  
.  
.  
.  
J is increased to 999.  
Write I WILL BE HONEST.  
J is increased to 1000.  
Write I WILL BE HONEST.
```

J is increased to 1001; since the value of J exceeds the final value, go on to the execution of subsequent tasks.

A question that might occur to you in studying this example is: What is meant by “subsequent tasks”? By this we simply mean the statements following the task to be repeated. Thus, in the following complete program, a thousand lines of I WILL BE HONEST will be printed, followed by the line THE END.

EXAMPLE 6-1

```
PROGRAM WRITEHONEST (OUTPUT);  
VAR J: INTEGER;  
BEGIN  
    FOR J:=1 TO 1000 DO  
        Writeln('I WILL BE HONEST');  
        Writeln('THE END')  
    END.
```

What happens is that after writing I WILL BE HONEST 1000 times, the computer proceeds to carry out the next task after the FOR statement—in this case, Writeln(‘THE END’).

Actually, the initial value of the index variable is not restricted to 1. The following two FOR statements will also print out the line I WILL BE HONEST 1000 times:

```
FOR J:=7 TO 1006 DO  
    Writeln('I WILL BE HONEST')  
  
FOR J:=0 TO 999 DO  
    Writeln('I WILL BE HONEST')
```

In Pascal, one is also allowed to count backwards:

```
FOR J:=1000 DOWNT0 1 DO  
    Writeln('I WILL BE HONEST')
```

In this case the index J will be set initially to 1000, and after each repetition of the given task, J will be *decreased* by 1. Thus the computer is counting *down*. When the value of J becomes *less* than the specified final value (in this case 1), the computer will go on to execute the next statement. (Caution: Notice that DOWNT0 is all one word in Pascal. Don’t put a space after DOWN!) The ability to count backwards does not, at first glance, seem to add much to what we can instruct a computer to do. However, we will later see some examples where this flexibility is useful.

A more obviously useful feature is the ability to specify the initial and final values of the index as integer variables instead of constants. (Your

physical education teacher might, for example, assign you a variable number of push-ups, where the value of the variable might be the number of errors you committed in yesterday's softball game.) Among other things, this feature allows programs to be written in a more general manner: for example, the number of repetitions might be supplied by reading it in or by computing it. If we haven't decided yet how many times we will make the computer write I WILL BE HONEST, we could write the following program:

```
PROGRAM WRITEHONEST ( INPUT , OUTPUT );
VAR J , FINAL : INTEGER;
BEGIN
    READ(FINAL);
    FOR J:=1 TO FINAL DO
        WRITELN ( 'I WILL BE HONEST' );
        WRITELN ( 'THE END' )
    END.
```

Or if we want to make the amount of punishment dependent on an input variable TIMESCHEATED, we might write:

EXAMPLE 6-2

```
PROGRAM WRITEHONEST( INPUT , OUTPUT );
VAR J , FINAL , TIMESCHEATED : INTEGER;
BEGIN
    READ(TIMESCHEATED);
    FINAL := 100 * TIMESCHEATED;
    FOR J := 1 TO FINAL DO
        WRITELN ( 'I WILL BE HONEST' );
        WRITELN ( 'THE END' )
    END.
```

The task to be carried out within a FOR statement can involve the value of the index, as illustrated in the following example.

EXAMPLE 6-3

```
PROGRAM LOOK INDEX ( OUTPUT );
VAR I : INTEGER;
BEGIN
    FOR I := 1 TO 10 DO
        WRITELN ( 'I = ' , I : 3 )
    END.
```

Let us look closely at what happens as this simple program is executed. First, the value of I is set to 1. Then I = 1 is printed. Then I is increased to 2, and the

test for whether I is greater than 10 (the final value) is performed. Since I is not greater than 10, the task (writing the value of I) is repeated. This continues until I does become greater than 10, so that the complete output from this program is

```
I = 1
I = 2
I = 3
I = 4
I = 5
I = 6
I = 7
I = 8
I = 9
I = 10
```

6-3 COMPOUND STATEMENTS

In general, a FOR loop (i.e., a loop in which a FOR statement is used to control the task repetition) is used to instruct a computer to repeat a task so many times. We then need to be able to specify precisely what the task is that is to be repeated. In the examples of the preceding section, the tasks were simple ones that could be specified by a single statement. But often we wish to repeat computational tasks that are too complicated to be specified in one statement. In Pascal, we can easily repeat a group of statements by a FOR loop. All we have to do is preface the group with BEGIN and end the group with END. The following example shows how this works.

EXAMPLE 6-4

```
FOR I := 1 TO 10 DO
  BEGIN
    J := I * I;
    K := J * J;
    WRITELN( I, J, K )
  END
```

In this example we have a single FOR statement. The task to be executed on each passage through the loop is specified by three statements (two assignments and a WRITELN), executed in sequence. As this FOR statement is executed, here is what happens. First the value of the index I is set to 1. Then the expression $I * I$ is computed and assigned to the variable J. Next, $J * J$ is computed and assigned to the variable K. Then I, J, and K will be printed as

```
1      1      1
```

Subsequently I will be incremented, tested to see whether it is greater than 10, and a second passage through the loop will result in the output:

2 4 16

As an exercise, continue this analysis of what happens as this FOR statement is executed. What is the complete output?

It is useful in many other situations to be able to define groups of statements and use them in place of single statements. In Pascal, a *compound statement* is defined to be any block of code that looks like

```
BEGIN
    [statements]
END
```

Since a compound statement is just another statement, it can be used any place in your program where a single statement is allowed. In particular, this means that the statements within a compound statement may themselves be compound. Furthermore, if you look at any of the complete programs given so far, you will notice that the statements are prefaced by BEGIN and followed by END. Hence these simple programs consist of one large compound statement.

So that you will clearly understand how compound statements may be contained within other compound statements, here's Example 6-4 expanded to a complete program.

EXAMPLE 6-5

```
PROGRAM IFOURTH(OUTPUT);
VAR I, J, K : INTEGER;
BEGIN
    WRITELN('SQUARES AND FOURTH POWERS');
    FOR I := 1 TO 10 DO
        BEGIN
            J := I*I;
            K := J*J;
            WRITELN(I, J, K)
        END
    END
END.
```

Notice that there are two BEGINS and two ENDS in this program. The first BEGIN indicates where the computation starts; i.e., where the set of statements to be executed begins. The second END indicates where the whole program ends. Together they form a BEGIN..END pair that brackets the statements comprising the executable part of the program. The other BEGIN..

END pair (the second BEGIN and the first END) brackets the set of statements that are to be repeated in the FOR loop.

As a last example in this section, we note a situation in which it is natural to use DOWNTO. Suppose we want to print a table of financial data for a company in reverse chronological order; i.e., beginning with the most recent year and proceeding back to earlier years. The program to do this might include a loop that looks like the following:

```
FOR YEAR := 1980 DOWNTO 1965 DO
  BEGIN
    WRITELN ( 'DATA FOR YEAR', YEAR:5 );

    (* Insert instructions to write
       financial data here *)

  END
```

6-4 SOME SUBTLE POINTS ABOUT FOR LOOPS

In previous examples, you have seen that the value of the index of a FOR loop can be printed out or used in computations inside the loop. It appears to act like an ordinary variable, but one which is automatically incremented and tested. However, an index is *not* like an ordinary variable in other respects. In particular, after the execution of a FOR loop is completed, the value of the index is "undefined," i.e., can be any number at all, depending upon how your local implementation works. We ran the following variation of Example 6-3 on our machine to see what would happen.

EXAMPLE 6-6

```
PROGRAM INDEXTEST ( INPUT, OUTPUT );
VAR I : INTEGER;
BEGIN
  FOR I := 1 TO 10 DO
    WRITELN ( 'I = ', I );
    WRITELN ( 'FINAL I = ', I )
  END.
```

The output of this program was

```
I =      1
I =      2
I =      3
I =      4
I =      5
```

```

I = 6
I = 7
I = 8
I = 9
I = 10
FINAL I = -576460752303423487

```

Clearly, you would cause some bizarre results if you proceeded to compute with this final I value, thinking it would be 11!

However, since the index looks like an ordinary variable *inside* the loop, can statements be included in the loop to explicitly change the value of the index—or even to read in a new value for the index? Clearly changing the index inside the loop would be a dangerous and confusing thing to do. Consider the following FOR statement

```

FOR I := 1 TO 10 DO
  BEGIN
    WRITELN('I = ', I);
    I := 2
  END

```

If this is legal, think what will happen. First, I is set to 1 and I = 1 is printed. Then I is set to 2, after which I is incremented and tested. On the second pass through the loop, I = 3 is printed. Then I is set to 2 again, etc. If left to itself, the computer would print successive lines of I = 3 forever! Such a repetitive task that never ends is called an *infinite loop*. Of course, a computer system will really not let a job run forever but will abort it after a (possibly long) time.

Because of the danger of infinite loops and the general confusion caused by allowing both the automatic index changes and the changes you as the programmer write, the proposed standard for the Pascal language specifies that any attempt to change an index value by the statement to be repeated shall cause an error. (That is, you should get an error message and your program should not run.) However, some implementations violate this rule and allow the index to be changed inside the loop. As an experiment, we tried the following program:

EXAMPLE 6-7

```

PROGRAM INDEXTEST(INPUT, OUTPUT);
VAR I: INTEGER;
BEGIN
  FOR I := 1 TO 10 DO
    BEGIN
      WRITELN('I = ', I);
      READ(I)
    END
  END.

```

3	6	9	12
---	---	---	----

and the output was

I	=	1
I	=	4
I	=	7
I	=	10

Follow through the execution of this loop so that you understand what happened. Was the last data item (12) read in? Run this example on your computer to see if it allows this sort of very poor programming practice. If it does, remember not to write programs like this anyway!

Another way to cause confusion in FOR loops would be to change the final test value inside the loop. For example, what happens when the following loop executes?

```

FINAL := 3;
FOR I := 1 TO FINAL DO
  BEGIN
    FINAL := FINAL + 1;
    Writeln (I, FINAL)
  END

```

Does the final value keep on increasing so that I never catches up? No, Pascal fortunately guards against this particular kind of disaster. What happens is that before the loop begins, the initial and final values are stored in special temporary memory locations for safekeeping. It is these temporary locations that are used to initialize the index and test for the end. The fact that inside the loop you change the value of the variable that was used initially to specify the final value will have no effect whatsoever on how many times the task is repeated. The output of the loop above is then

1	4
2	5
3	6

Another point you might have wondered about is what happens if the final value for the index is less than the initial value, as in:

```

FOR I := 10 TO 5 DO
  Writeln ('IS ANYTHING WRITTEN?')

```

To explain this point, we must look more closely at what happens when a FOR loop is executed. (We have simplified our previous descriptions, in order not to complicate matters too much at first.) When a FOR loop is executed, the

index variable is initialized and then is *immediately* compared with the final value. So whenever the initial value is greater than the final value, the statement within the loop is never executed. Thus, in this example, the first step is to set $I = 10$. Then the test is made. (Is $I > 5$?) Since the test is satisfied the statement to be repeated is executed zero times.

Why might we want to be able to skip over a loop entirely? Here's a simple example. Suppose that we want to give a salesperson a bonus for meeting his or her quota and for each item sold over that quota. We might write

```
FOR I := QUOTA TO TOTALSALES DO
  WRITELN ( 'GIVE THAT PERSON A BONUS' )
```

Then if a salesperson doesn't meet the quota ($TOTALSALES < QUOTA$), not a single one of the magic lines will be printed.

Everything works analogously, of course, when the index is counted down instead of up. For example, the statement

```
FOR I := 1 DOWNT0 2 DO
  WRITELN( ' IS ANYTHING WRITTEN?' )
```

will cause nothing to be written, since immediately after setting I to 1 the termination test (Is $I < 2$?) is carried out.

In Pascal, the initial and final values of loop indexes may be specified not only by constants or variables but by expressions. Thus we could simplify Example 6-2 as follows:

EXAMPLE 6-8

```
PROGRAM WRITEHONEST( INPUT , OUTPUT );
VAR J , TIMESCHEATED : INTEGER;
BEGIN
  READ( TIMESCHEATED );
  FOR J := 1 TO 2 * TIMESCHEATED DO
    WRITELN( 'I WILL BE HONEST' );
  WRITELN( 'THE END' )
END .
```

Compare this program carefully with Example 6-2. By using an expression ($2 * TIMESCHEATED$) as the final value, we have eliminated the need to define the extra variable *FINAL* and assign a value to it. The compiler automatically allocates a memory location to hold the final value and stores $2 * TIMESCHEATED$ in that location as the loop is set up.

The capability of using expressions as limits in FOR statements raises another question. What happens if the expression does not yield a value of type *INTEGER*? You might think that there would be no problem with such a loop, that it would simply stop when the counter exceeds the noninteger value. But according to the Pascal language definition, the index limits

should be INTEGER variables or expressions; therefore, in all probability your compiler will flag a type error and your program will not be executed. If you really need to compute a loop limit by an expression that doesn't yield an INTEGER result, use TRUNC or ROUND to convert the result to an integer. And don't forget that ROUND may cause one more execution of the task than TRUNC will. (Why?)

Before leaving this section, let us note that it frequently is more natural to think of the limits of a FOR loop as REAL values than as INTEGER values and to want the increment to be some value other than 1. We must then somehow accommodate the natural definition of the problem to the Pascal restrictions. Here's an example. Suppose we want to compute and print a currency conversion table, converting English pounds into their equivalent in U.S. dollars. We might want to do this conversion for values of the variable POUNDS from 0.25 to 10.0, incrementing the value of POUNDS by 0.25 on each pass of the loop. This is easily accomplished by writing

```
FOR I := 1 TO 40 DO
  BEGIN
    POUNDS := 0.25 * I;

    (* Do conversion and print result *)

  END
```

In general, when faced with such a problem we simply need to find an arithmetic expression ($0.25 * I$ in this example) that will convert the consecutive integer values into the desired sequence of variable values.

6-5 FLOWCHARTS

Up until this chapter, all the programs we looked at consisted of a set of simple statements to be executed one by one in sequential order. It was then quite easy for someone to look at one of our programs and understand what the program was doing. Now that we have introduced loops and compound statements, it begins to be easy to get confused. (Instead of telling a little boy to go directly from home to the supermarket, then to the newsstand, then to the post office, you tell him to go from home to the supermarket, then come home; go back to the supermarket, then come home again; go back to the supermarket, then come home again; and then go to the newsstand and then to the post office. No wonder he gets confused!) We need a way to give an overall description of the major computational steps in a program without getting into all the syntactic details. Some such specification is enormously useful to a programmer. When asked to write a program to carry out a complex task, a programmer, instead of plunging at once into all the details of the program, should follow good engineering practice and lay out first in broad brush strokes the major steps of the computation, and then fill in the details later.

Even after a program has been completely written, it is important that a broad specification exist. Programs often need improvements or other changes. (This is known as *program maintenance*.) For example, a change in the income tax deduction rules will require changes in every existing payroll-computation program. The job of making these changes may be assigned to the original programmer (who probably doesn't remember very well what he or she did) or, more likely, to another programmer who knows nothing about the program. In either case, there is a critical need for an understandable overview of the program—its various parts, what each is designed to do, and how they work together to get the job accomplished.

There are various ways that such a specification could be made. One way is just by a verbal description. Unfortunately, as all of us have discovered, it is not easy to make a verbal description of a complicated task clear, complete, and unambiguous. (For example, you have probably found the descriptions of the machine problems in this text not quite as clear as you might wish them to be.)

What is needed is a more formal, structured way to specify what a program should do. One such way is to use a *flowchart*, or formal diagram showing what is to be accomplished in each part of the program and the relationship of these parts to one another. Several different styles of flowcharts have been developed over the years. The modular style we will present here is a rather recent development, designed to be more compatible with modern programming languages such as Pascal.

In the flowchart, a task to be carried out is represented by a rectangle. Thus, a set of tasks to be executed in succession is charted as a sequence of rectangles. For example, Figure 6-1 shows a flowchart that could serve as a broad specification for many different programs, depending on the details of the "computation."

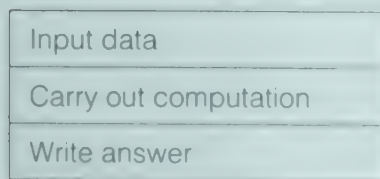


Figure 6-1

There are two interesting points to notice here. First, the three successive major tasks, each itself represented by a rectangle, are all contained in a single large rectangle. This makes sense. We think of the whole program as a single large task that is broken down into three major subtasks. Second, notice that the middle block in this flowchart is vague, to say the least. Precise specification of this step will involve subdividing this rectangle into smaller rectangles (steps involved in the computation) and perhaps introducing FOR loops or other constructs.

A FOR loop (or any of the other kinds of loops that will be introduced later) is indicated in the flowchart as shown in Figure 6-2.

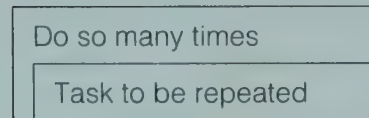


Figure 6-2

Again, notice that the whole loop specification forms a single rectangle (it can, after all, be written as a single FOR statement) and nested within is a second rectangle to specify the task to be repeated.

A nice thing about this type of flowcharting is that there is not very much to learn. There will be one or two other types of pictures or diagrams that we will introduce later as needed. But just this much knowledge will allow you to draw and understand flowcharts for quite complex programs, as you will see.

As a very simple example, Figure 6-3 shows a flowchart for Example 6-5.

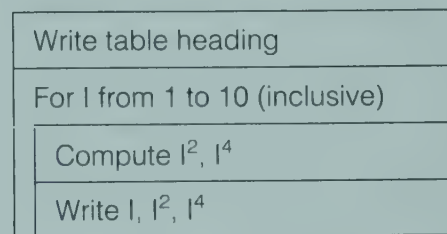


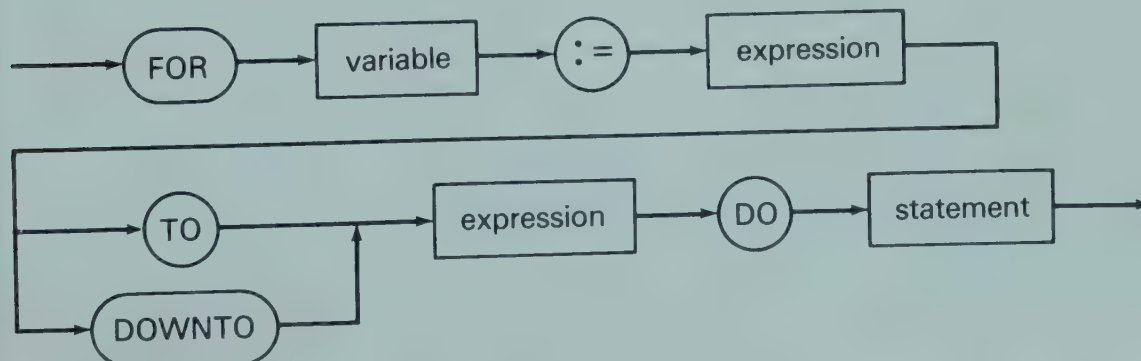
Figure 6-3

Notice that the loop here is not headed by "Do 10 times." That's because the index I is involved in the computation within the loop, and we needed to introduce it and indicate that it is incremented on every passage through the loop. The point is that the statements one puts in the flowchart are English, not Pascal, and one is allowed some flexibility in stating in a convenient way what's to be done. At the same time, of course, it is important that those statements be unambiguous, so that there is no difficulty in translating the flowchart into a Pascal program.

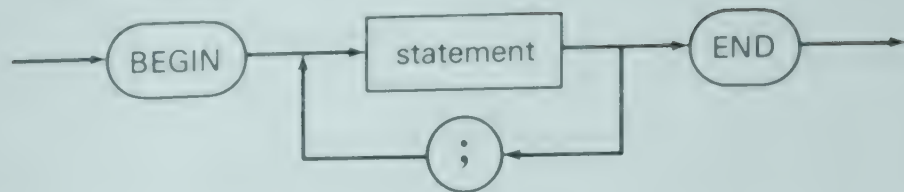
6-6 SYNTAX

There are only two new syntax diagrams needed to formalize the constructions described in this chapter.

First, the *FOR statement*:



Second, the *compound statement*:



Notice carefully the punctuation here. As usual, semicolons are used to *separate* statements, in this case the individual statements that go to make up the compound statement. No semicolon is needed after the last statement, however, since the END provides an adequate indication of where the statement ends. (A feature of Pascal that you may discover to be missing in other programming languages is that punctuation is minimal, generally only being required when it is needed by the compiler to figure out what you mean.)

6-7 NOTES ON PROGRAMMING STYLE AND PITFALLS

Once you know how to program a loop, you have the ability to use up (and possibly waste) enormous amounts of computer time.

Consider the following loop.

```

FOR I:=1 TO 100 DO
  BEGIN
    READ(RAD);
    CIR:=2.0 * 3.1416 * RAD;
    Writeln (RAD, CIR)
  END

```

Here a simple task is repeated 100 times, namely, reading in the radius of a circle and computing the circumference of that circle. This is a correct program segment, but there is one point that can be improved upon. In repeating the FOR loop 100 times, the computer will multiply $2 * 3.1416$ a hundred times. This is clearly wasteful. To make this piece of code more efficient, move the multiplication of the constants *outside* the loop, as in the following program.

```

PROGRAM CIRCLES (INPUT, OUTPUT);
(* COMPUTES CIRCUMFERENCES OF CIRCLES *)
VAR TWOPI, RAD, CIR : REAL;
    I : INTEGER;
BEGIN
  TWOPI := 2.0 * 3.1416;
  FOR I := 1 TO 100 DO
    BEGIN (* LOOP *)

```

```

        READ (RAD);
        CIR := TWOPI * RAD;
        WRITELN (RAD, CIR)
    END (* LOOP *)
END .

```

This example illustrates a general principle of efficient programming: *Do not include any computation inside a loop which can just as well be done outside the loop.*

If you look back to Section 6-2, you will notice that different ways were given there for specifying that a task be repeated 1000 times. Probably you found the first one the most natural and easiest to understand. If so, *that is the one you should use.* We pointed out the other possibilities mainly to give you a more complete idea of the syntax of the FOR statement. We don't recommend, for example, that a task to be repeated 55 times be controlled by a counter that runs from 27 to 81. The possibility of your making a mistake in the arithmetic is obvious; it is perhaps less obvious that anyone trying to understand your program will be irritated and maybe even confused by such unnecessary complexities. Another important principle of programming is: *Keep things simple and do tasks in a standard way. Don't look for novel ways to do simple tasks!*

Finally, let us point out the possibilities for expensive errors that exist because the initial and final values for the index in a FOR loop can be variables or even expressions. Suppose you were to write the following piece of code:

```

    READLN (K, L, M);
    FOR J := 1 TO K * L - M DO
        WRITELN ('WRITE LOOPS CAREFULLY')
    
```

If the data values read in are 1000, 2, 1000, the loop will obviously be executed 1000 times. But what if a careless typist types the input data in the wrong order, say 1000, 1000, 2? This would generate a stack of paper about 6 feet high!

6-8 EXERCISES

6-1. What will the following program segment print?

```

    FOR J := 1 TO 3 DO
        WRITELN(' I WILL');
        WRITELN(' BE HONEST')
    
```

6-2. What is the value of T after each of the following program segments is executed?

```

(a) T := 0;
    FOR K := 10 DOWNT0 1 DO
        T := T + K
    
```



```
(b) T := 10;
    FOR K := 1 TO T DO
        T := T + K
```

6-3. What is printed by the following program segments?

```
(a) J := 6;
    FOR I := 1 TO 5 DO
        BEGIN
            J := J - I;
            WRITELN(I, J)
        END
```

```
(b) J := 14;
    FOR I := 5 DOWNT0 1 DO
        BEGIN
            WRITELN(I, J);
            J := J - I
        END
```

6-4. Draw a flowchart for Example 6-8.

6-5. Rewrite the program segment below so that it is more efficient. (Use any additional variables you may require.)

```
READLN(X, PI);
FOR R := 1 TO 100 DO
    BEGIN
        V := (X*PI*((R*2) * (R*2) * (R*2))) / 3;
        WRITELN(V)
    END
```

6-6. Write a Pascal program to produce the following output:

```
I WILL BE HONEST 1 TIME.
I WILL BE HONEST 2 TIMES.
I WILL BE HONEST 3 TIMES.
.
.
.
I WILL BE HONEST 50 TIMES.
```

Construct a flowchart for your solution *before* writing any Pascal statements.

6-7. Write a Pascal program that reads in an employee's number, hourly wage, and hours worked; then computes total pay earned and prints a message such as:

```
EMPLOYEE 4176 IS TO BE PAID $226.90
```

Keep a running total of the amount owed all employees and, after all data is read in, print the total payroll.

(The first value on the first data card is the number of employees for which this computation is to be performed.)

Sample data

3			First card (or line)
4176	5.02	45.2	
5011	7.45	40.2	
9055	6.80	31.6	Last card (or line)

Construct a flowchart for your solution *before* writing any Pascal statements.

- 6-8.** Assume that an addition takes two units of time, an assignment of a value to a location takes one unit of time, and a multiplication takes 10 units of time. How long does the following program segment take to execute? (Ignore the loop “overhead”; i.e., the time it takes the computer to initialize I, increment it, and test whether enough repetitions have been completed.)

```

K := 14;
FOR I := 1 TO 20 DO
    BEGIN
        ISQ := I * I;
        VALUE := ISQ * K + VALUE
    END
    
```

- 6-9.** Write a Pascal program that reads an integer N and computes:
- (a) The sum of the integers from 1 to N
 - (b) The sum of the squares of the integers from 1 to N

SUMMARY OF CHAPTER 6

The FOR statement is used to repeat a task a prescribed number of times. The task to be repeated is specified by a statement that may be a compound statement (delineated by the key words BEGIN and END).

The index in a FOR statement can be stepped up (LOWER to UPPER) or stepped down (UPPER to LOWER). The increment or decrement of the index in each step is always 1.



ARRAYS

7-1 INTRODUCTION

Probably you have heard of that famous private detective from the orient, Mr. Charlie Chan—maybe you have seen him on the late late show. Mr. Chan had several sons whom he referred to as “number 1 son,” “number 2 son,” and so on. Now most parents attack the difficult problem of “naming the baby” by a variety of time-consuming techniques. Some ponder names they like; others consult friends and relatives; still others (growing more desperate) dig into their local libraries for books such as *A Thousand Most Beautiful Names for Baby Girls*. Contrast Mr. Chan’s wise and farseeing solution. Suppose he had 17 sons. Not only did Mr. Chan not have to spend the time and effort to choose a name each time a son was born, but also he did not need to worry about using a name that is readily distinguished from one already used. Consider another convenience of identifying children by number. When a family has 17 sons, the parents might just possibly be able to remember the order in which they arrived. But relatives and friends are bound to be confused as to whether Paul is older than David or whether Anthony is the third oldest son in the family. In Mr. Chan’s household there will be no such confusion, since number 3 son is clearly older than number 7 son, and number 3 son is clearly the third oldest son in the family. Mr. Chan’s wisdom can further be appreciated in other situations: Instead of instructing Paul, Edward, Jimmy, Danny, Robert, and William to give an eyewitness account of an accident, Mr. Chan can simply say that sons number 4 through 9 must do so. Mr. Chan can also invite number 3 son and number 3 daughter-in-law for dinner without taxing his memory as to the name of the wife of number 3 son.

At this point you might be beginning to wonder what all this has to do with computer programming. The point is that when writing a program we need to use different names for different variables. Often we have to deal with large sets of similar variables: the salaries of all the employees of a company or the grades of all the students in a class. The lesson to be learned from Mr. Chan is just this: Instead of thinking up a new name for each salary (JOESSAL, BOBSSAL, etc.) we might refer to the salaries in the list as number 1 salary, number 2 salary, etc. This is rather awkward, of course, so what we really want to do is use a single variable name with a *subscript* to distinguish between the individual salaries. You should have seen variables with subscripts (or *subscripted variables*) in high school algebra; for example, pairs of equations are commonly written

$$a_1x + b_1y = c_1$$

$$a_2x + b_2y = c_2$$

In this example, the subscripts distinguish the coefficients in the two equations. Unfortunately, writing subscripts a half-line down is not practical in communication with a computer. Whether you are typing your program on punched cards or using a terminal, everything naturally comes out on one line. Instead, square brackets are used in Pascal to denote subscripts, e.g., SALARY[1], SALARY[2], etc. More formally, a *subscripted variable* in Pascal is a name (identifier) followed by a *subscript*; i.e., an integer (or integer-valued variable or expression) enclosed in square brackets. A fixed number of such variables, all of the same type and referenced by the same name (followed by a subscript), is called an *array*. Sometimes the name (without a subscript) is used to refer to the whole array collectively. Thus we might have an array named SALARY consisting of the variables

SALARY[1], SALARY[2], . . . , SALARY[100]

At this point, you might ask why we do not simply use names like SALARY1, SALARY2, . . . , without the complexity of adding brackets to set off a part of the name as a subscript. We shall answer this question in the next section.

7-2 USE OF SUBSCRIPTED VARIABLES

Subscripted variables can be used in the same way as unsubscripted variables. For example, subscripted variables are used in the following program segment:

```
READ (X[1], X[2]);  
X[3] := X[1] + X[2];  
X[4] := X[1] * X[2];  
WRITELN (X[3], X[4])
```

Looking at just these four statements by themselves, we see that they perform the same computation as the statements

```
READ (A, B);  
C := A + B;  
D := A * B;  
WRITELN (C, D)
```

However, adding subscripts to variables is more than just a convenient way to create new variable names. To explain the advantages of using subscripted variables, let us note first that the subscript of a variable is not restricted to be an integer constant. The subscript can be, for example, an integer variable. That is, we can write $X[I]$ where I is an integer variable. Consider the following program segment:

```
READ (I);  
RATE := 1.05 * X[I];  
WRITELN ('RATE FOR I =', I, ' IS', RATE:6:2)
```

If the value of I is read in as 7 then $RATE$ is computed just as if we had written the statement

```
RATE := 1.05 * X[7]
```

On the other hand, if the value of I is read in as 18, then $RATE$ is computed as

```
RATE := 1.05 * X[18]
```

If we want to compute and write the rate for a whole array of values $X[1]$, $X[2]$, ..., $X[20]$, we can use a loop:

```
FOR I := 1 TO 20 DO  
  BEGIN  
    RATE := 1.05 * X[I];  
    WRITELN (I, X[I], RATE)  
  END
```

Notice that the index variable in the `FOR` statement is also used as the subscript. Then as the index is incremented on successive executions of the statements in the loop, the rate computation is carried out with the successive values $X[1]$, $X[2]$, ..., $X[20]$. Now you see the real power of subscripts. If we tried to do this same computation with a list of 20 variables named $X1$, $X2$, ..., $X20$, we would not be able to use a loop, but would have to write down the computation separately for each variable in the list.

Here's another example. Suppose we have a deck of 100 cards, each containing the grade of a student. The following program segment reads these grades in, stores them in $Grade[1]$, $Grade[2]$, ..., $Grade[100]$, and

prints out the 10 grades GRADE[91] through GRADE[100] together with their sum.

EXAMPLE 7-1

```
FOR I := 1 TO 100 DO
    READ (GRADE[I]);
SUM := 0;
FOR I := 91 TO 100 DO
    BEGIN
        WRITELN (GRADE[I]);
        SUM := SUM + GRADE[I]
    END;
WRITELN (SUM)
```

Note that in both FOR loops the index variable is used as a subscript. Consequently the program will read in the values of GRADE[1] through GRADE[100] in the first FOR loop and will compute the sum of GRADE[91] through GRADE[100] in the second FOR loop. Incidentally, there is no need to use the same name for the index variable in both of these loops. For example, we could just as well read in the grades with the statement

```
FOR INDEXVALUE := 1 TO 100 DO
    READ (GRADE[INDEXVALUE])
```

What matters is not the name we give the index (and subscript) but the set of *values* that variable takes on as the loop is executed. So there's no reason why we need to pick the same name for different loops (unless it helps to make the program more easily readable). Furthermore, although we tend to use simple names like I for index variables and subscripts, you might want to remember the lesson of Section 3-6-1 and use more descriptive names. For example, we could rewrite Example 7-1 as

```
FOR STUDENTNUMBER := 1 TO 100 DO
    READ (GRADE[STUDENTNUMBER]);
SUM := 0;
FOR GRADENUMBER := 91 TO 100 DO
    BEGIN
        WRITELN (GRADE[GRADENUMBER]);
        SUM := SUM + GRADE[GRADENUMBER]
    END;
WRITELN (SUM)
```

There is another important lesson to be learned from Example 7-1. Notice the statement just preceding the loop in which numbers are added to sum; this statement assigns the value zero to SUM. Thus we have *initialized*

the variable *SUM* to the value zero. Without such explicit initialization, we won't know what the number initially in location *SUM* might be (perhaps it is something left over from some other computation), and we will be computing the sum we want *plus* the initial unknown value of *SUM*.

Subscripts are not restricted to integer constants or integer variables, but can be any arithmetic expression whose value is an integer. Suppose we have a deck of 200 cards, the first 100 cards containing the grades in the first hourly examination for 100 students and the next 100 cards containing the grades in the second hourly examination. The following program segment will compute the total grade of each student. (We assume that the grades for the two hourly examinations are arranged in the same order.)

EXAMPLE 7-2

```
FOR I := 1 TO 200 DO
  READ (GRADE[I]);
FOR I := 1 TO 100 DO
  F[I] := GRADE[I] + GRADE[I + 100]
```

Then to print out every fifth total grade; namely, *F*[1], *F*[6], *F*[11],...,*F*[96], we could write:

```
FOR I := 0 TO 19 DO
  WRITELN (F[5 * I + 1])
```

7-3 DECLARING ARRAYS

As we discussed earlier, each variable used in a Pascal program is allocated a memory location in which the value of the variable is stored. Therefore, when we use arrays we must make sure that every subscripted variable in the array is assigned a memory location. This is done by declaring the number of memory locations required for the array. For example, the declaration

```
VAR X:ARRAY [1..50] OF INTEGER;
```

declares that *X* is an array containing the subscripted variables *X*[1], *X*[2],..., *X*[50]. Consequently, 50 memory locations will be reserved for *X*[1], *X*[2],..., *X*[50]. Furthermore, by declaring *X* to be an array OF INTEGER we also have declared that *X*[1], *X*[2],...,*X*[50] are all integer variables. (Remember that all the subscripted variables in a single array must belong to the same data type. This is not a severe restriction because the whole idea behind arrays was the desire to be able to name and use sets of similar variables conveniently.)

Note that in the example above, we declare that the legal values for the subscript are the integers 1 through 50, inclusive. The smallest value that can be taken on is referred to as the *lower bound* on the subscript values—in

this case the lower bound is 1. Similarly, the largest value that can be taken on is referred to as the *upper bound* on the subscript values—in this case the upper bound is 50. It is illegal to use subscript values outside the given bounds. For example, since there is no memory location allocated to $X[51]$, we cannot assign a value to it or try to use its (undefined) value in a computation.

Just as the initial value of the index variable in a FOR loop need not be 1, the lower bound on a subscript need not be 1. For example, the declaration

```
VAR Y:ARRAY [1985..1988] OF REAL;
```

specifies that the array Y contains four subscripted variables $Y[1985]$, $Y[1986]$, $Y[1987]$, and $Y[1988]$. Consider how we might use this facility. Suppose we have a Pascal program in which we compute the projected profits of a company for the years 1985, 1986, 1987, and 1988. It would be convenient to refer to these quantities as $Y[1985]$, $Y[1986]$, $Y[1987]$, and $Y[1988]$ instead of $Y[1]$, $Y[2]$, $Y[3]$, and $Y[4]$. On the other hand, it would be wasteful to reserve space for 1988 REAL variables by using the declaration

```
VAR Y:ARRAY[1..1988] OF REAL;
```

As a matter of fact, a negative integer value can also be used as a subscript. Thus, the declaration

```
VAR Z:ARRAY[-2..5] OF INTEGER;
```

specifies that Z is an array containing the eight subscripted INTEGER variables $Z[-2]$, $Z[-1]$, $Z[0]$, $Z[1]$, ..., $Z[5]$. For example, we might use $Z[0]$ to denote the number of workers whose productivity is at a normal level and $Z[-1]$, $Z[-2]$ to denote the numbers of those whose productivities are below normal and much below normal, respectively.

Let us emphasize that the lower and upper bounds of the subscript of an array must be integer *constants*, so that the number of memory locations to be reserved is specified explicitly. Thus, the following declaration is illegal:

```
VAR X:ARRAY [1..N] OF INTEGER;
```

because there is no way to tell what value N assumes. But this can be made legal by writing:

```
CONST N = 20;  
VAR X : ARRAY [1..N] OF INTEGER;
```

On the other hand, there is no way to read in the size of an array or assign a value to the size, since the memory locations for the array must be reserved before the program is executed. Thus if you plan to read different numbers of variables into an array at different times (e.g., the number of students taking a course will vary from semester to semester), you will want to declare an array to be of the maximum size anticipated.

We shall say more about the declaration of arrays in Chapter 14, after the notion of defining new data types has been discussed.

7-4 HIGHER DIMENSIONAL ARRAYS

Let us go back to our detective from the orient. We note that Mr. Chan will also have a rather systematic way to refer to his grandchildren. Clearly, a grandson can be referred to as, for example, number 3 son's number 7 son. This suggests the use of variables with two subscripts. In Pascal, a variable with two subscripts is written as $X[3,7]$. Furthermore, a *two-dimensional array* is a set of variables with two subscripts that share a common name and are all of the same data type. For example, suppose a company has 200 employees. For each employee, we need to keep a record of the number of hours worked each day during a seven-day period. To do this we can use a two-dimensional array HOUR in which, for example, the subscripted variable HOUR[27,5] will be the number of hours employee number 27 worked on the fifth day. The declaration of a two-dimensional array follows the same format as for a one-dimensional array, except that we must specify the lower and upper bounds of both subscripts. The declaration we need for this example is:

```
VAR HOUR: ARRAY[1..200, 1..7] OF INTEGER;
```

The whole array HOUR will contain 1400 variables: HOUR[1,1], HOUR[1,2], ..., HOUR[1,7]; HOUR[2,1], ..., HOUR[200,7]; i.e., for each of the values 1...200 of the first subscript, the second subscript will take on values 1...7. Or, in terms of this specific example, for each of the 200 employees, there are seven variables, which correspond to the hours worked on each of the seven days of the week.

Now suppose that the values of HOUR[I,J] have already been stored in this array by one part of a program. (We will discuss how to do this in the next section.) Then the following program segment will read in a value of the subscript I (i.e., an employee identification number), compute the total number of hours worked by that employee, and write the result.

EXAMPLE 7-3

```
READ(I);  
TOTAL := 0;  
FOR J := 1 TO 7 DO  
    TOTAL := TOTAL + HOUR[I,J];  
WRITELN(I, TOTAL)
```

Arrays of dimension 3, 4, 5, etc., can also be constructed. You just need to increase the number of subscripts and declare their bounds. (See the syntax diagrams of Section 7.6.) Suppose, for example, you want to create an array

which holds the number of hours worked each day by each of 200 employees, for each week of the year. This means adding an extra dimension to our array HOUR. The declaration might look like:

```
VAR HOUR: ARRAY[1..200, 1..7, 1..52] OF INTEGER;
```

HOUR[100,2,10] would now refer to the number of hours worked by employee number 100 on the second day of the tenth week. What is the total number of variables that is contained in this array?

7-5 NESTED FOR LOOPS

Let us expand a bit on Example 7-3. In particular, let us write a program segment that will compute for every employee the total number of hours worked in the week. That is, for each value of the index I we want to carry out the computation of Example 7-3. Clearly, instead of reading in a particular value of I, we want to use a FOR statement with control variable I to repeat the computation of TOTAL for $I = 1, 2, \dots, 200$. Here's a piece of code that will do the job.

EXAMPLE 7-4

```
FOR I := 1 TO 200 DO
  BEGIN
    TOTAL := 0;
    FOR J := 1 TO 7 DO
      TOTAL := TOTAL + HOUR[I, J];
    Writeln (I, TOTAL)
  END
```

Notice that there are two FOR loops in this program, one *nested* inside the other. The outer FOR loop, which uses the control variable I, will cause the entire task specified between the BEGIN and END to be repeated 200 times. The inner FOR loop is a part of the task repeated under the control of the outer FOR statement. What happens is then just what we desire: the whole inner loop—the computation, itself requiring repetition, that computes the total hours worked by employee I—is repeated 200 times, one for each value of I.

The structure of Example 7-4 becomes particularly clear if you look at its flowchart, shown in Figure 7-1. Notice how visible the nesting is in this chart. This displays at a glance how the inner loop (Figure 7-2) is part of the task which comprises the outer loop.

It is also worthwhile to look in closer detail at what happens—in particular at how the indexes are incremented—as this program is executed. The steps are:

- Set I to 1.
- Set TOTAL to 0.
- Set J to 1.

Add HOUR[1,1] to TOTAL.
 Increase J to 2.
 Add HOUR[1,2] to TOTAL.
 Increase J to 3.
 Add HOUR[1,3] to TOTAL.

.....
 Increase J to 7.
 Add HOUR[1,7] to TOTAL.
 End inner loop.
 Output result.
 Increase I to 2.
 Set TOTAL to 0.
 Set J to 1.
 Add HOUR[2,1] to TOTAL.
 Increase J to 2.
 Add HOUR[2,2] to TOTAL.

And so forth. Notice that we have been careful to remember to set TOTAL to zero before computing the hours worked for *each* employee. If we only initialized TOTAL once, we would continue to add the hours worked by successive employees on top of all the hours worked by the preceding employees. By the time we reached the 200th employee, TOTAL would become amazingly large!

There is one more lesson to be learned from Example 7-4. Notice that when we write programs with nested FOR loops, we indent the lines of code just as they naturally are indented when we draw the flowchart. In Pascal there are no rules requiring indentation or beginning program statements at particular line positions. You are free to format your program as you wish. It won't matter to the compiler if, for example, you begin all lines at the left margin. But it will make it harder for you—or anyone else—to figure out what

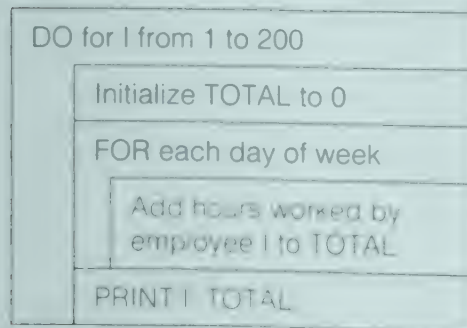


Figure 7-1

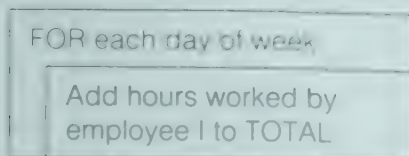


Figure 7-2

the program is supposed to accomplish. Indentation is invaluable in helping programmers keep track of where they are, as well as in making programs easier to read.

In Example 7-4 there is also a separate level of indentation for the BEGIN and END. This is because it is of critical importance not to get confused about where they go or to forget one. Consider the following piece of code that is a small variation on Example 7-4; the only difference (in structure) is that now the inner loop also executes a compound statement.

```
FOR I := 1 TO 200 DO
  BEGIN
    TOTAL := 0;
    FOR J := 1 TO 7 DO
      BEGIN
        WRITELN (HOUR[I,J]);
        TOTAL := TOTAL + HOUR[I,J]
      END;
    WRITELN (TOTAL)
  END
```

Suppose the second BEGIN were left out. The outer loop would then be assumed to end at the first END, and the inner loop would only contain the single WRITELN statement. One hopes, of course, that the compiler, noticing that the END that ends this segment has been rather left dangling, will then print an error message. But there's always the horrible possibility that this last (or some other) END has also been omitted. Then the syntax might look all right to the compiler, but the code that executes will not be exactly what the programmer had in mind.

As an aid to keeping the BEGINS and ENDS straight, we can explicitly include them in the flowchart. For example, see Figure 7-3.

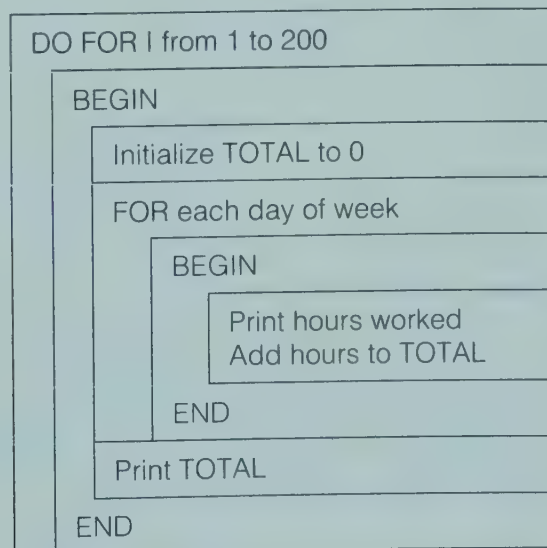
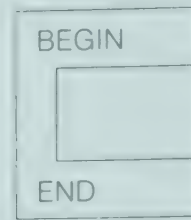


Figure 7-3

The new picture or symbol introduced into the flowchart here is very much like a bracket:



It plays the role of formally bracketing groups of statements into a compound statement and, incidentally, makes it hard for the programmer to forget a BEGIN or an END.

Before leaving this section, let us make good on our promise to explain how you can read the data values into the array HOUR. We first have to give a bit of thought to deciding on a convenient way to type the data. Each data item is an integer number of hours worked in one day, hence an integer between 0 and 24. Clearly we can fit seven such numbers on one data card (or line of input typed on a terminal). Assume then, that our input looks like:

0	8	8	8	8	8	0	(data for employee 1)
0	9	8	10	10	4	6	(data for employee 2)

and so forth, for 200 lines in all. Now each value will be read by a statement

```
READ ( HOUR[ I , J ] )
```

To read the seven values on each card, we need a loop

```
FOR J := 1 TO 7 DO
  READ ( HOUR[ I , J ] )
```

then to read all of the 200 data cards, we clearly must nest this loop inside another, giving

```
FOR I := 1 TO 200 DO
  FOR J := 1 TO 7 DO
    READ ( HOUR[ I , J ] )
```

Thus a single FOR statement (with another FOR nested inside it) suffices to read in all the data values for this entire array of 1400 variables.

Now let us put together the bits and pieces of programs discussed so far in this section into a complete program that (1) reads in the work data for the 200 employees and (2) prints a table (nicely formatted) of the total hours worked during the week by each employee.

```
PROGRAM TOTALHOURS (INPUT, OUTPUT);
(* READS WORK DATA AND PRINTS REPORT *)
VAR I,J,TOTAL : INTEGER;
    HOUR: ARRAY[1..200, 1..7] OF INTEGER;
BEGIN
    FOR I := 1 TO 200 DO (* FOR EACH WORKER *)
        FOR J := 1 TO 7 DO (* FOR EACH DAY *)
            READ (HOUR[I,J]);
        WRITELN ('WORKER', 'HOURS':8); (* HEADING *)
        FOR I := 1 TO 200 DO (* FOR EACH WORKER *)
            BEGIN (* TOTAL AND PRINT HOURS *)
                TOTAL := 0;
                FOR J := 1 TO 7 DO
                    TOTAL := TOTAL + HOUR[I,J];
                WRITELN (I:5, TOTAL:8)
            END
        END
    END.
```

Let us look at one more example before leaving this section. Suppose we are given a table of distances between cities. We can store the distances in an array DIST, where the value of DIST[I,J] is the distance from city I to city J. Now this array has some interesting properties. First, it is symmetric, that is, DIST[I,J] = DIST[J,I]. (We all know that the distance from St. Louis to Chicago is the same as the distance from Chicago to St. Louis.) Second, the values of the elements DIST[I,I] are zero, since the distance from a city to itself is zero. Now suppose someone asks us for the total distance he will travel if he travels between every pair of cities once. We could compute the sum by adding all the elements of the array and then dividing by two (since each distance appears twice). A program segment to do this might look like the following.

EXAMPLE 7-6

```
SUM := 0.0;
FOR I := 1 TO N DO
    FOR J := 1 TO N DO
        SUM := SUM + DIST[I,J];
    SUM := SUM/2.0
```

In this segment, N is the number of cities. If, say, there are 100 cities, this small piece of code would cause 10,000 additions. It is much more efficient to add in the distance between each pair of cities only once. Also, there is no point in adding in the zero values $\text{DIST}[I,I]$. Let us consider how we can decrease the number of additions. Note that the array DIST can be pictured as shown in Figure 7-4. The array is drawn there as N rows and N columns. The element at the intersection of row I and column J is $\text{DIST}[I,J]$.

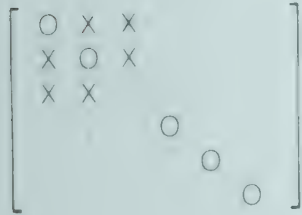


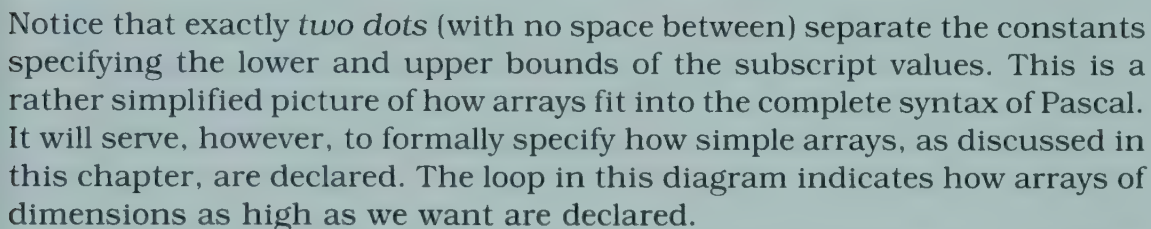
Figure 7.4

A two-dimensional array in this rectangular format is often referred to as a *matrix*. Notice the zeros along the diagonal from top left to lower right. (This is called the *main diagonal* of the matrix.) These zeros appear because the elements $\text{DIST}[I,I]$ are all zero. Now with a bit of thought you should be able to convince yourself that the elements $\text{DIST}[I,J]$ and $\text{DIST}[J,I]$, where $I \neq J$, lie on different sides of the main diagonal. This implies that to get the sum we want, we need only add the elements from one or the other of the triangular arrays into which the main diagonal divides this matrix. Suppose we add the elements of the lower-left triangle. We need to add one element from row 2, 2 from row 3, 3 from row 4, etc. Looking back at Example 7-6, we see that a critical change will have to be made. The final value of the control variable for the inner loop must be different every time it is executed. This is easy to do: since the initial and final values of loop-control variables may be any integer-valued expressions, they may even be expressions that involve the control variable of an outer loop. Thus we can rewrite Example 7-6 as:

EXAMPLE 7-7

```
SUM := 0.0;
FOR I := 2 TO N DO
    FOR J := 1 TO I-1 DO
        SUM := SUM + DIST[I,J]
```

How many additions are executed by this code segment? As an exercise, write a FOR statement to compute the sum of the elements in the upper-right triangle instead.



Now that we need to refer to subscripted variables as well as variables simply named by identifiers, we need a syntax diagram for *variable*.



No new syntax is needed to describe nested FOR loops. To convince yourself of this, just look at the syntax diagrams in the preceding chapter. A FOR statement causes repeated execution of a statement—which may itself be another FOR statement or perhaps a compound statement including one or more FOR statements. If you thoroughly understand this syntax, you are unlikely to make any careless mistakes in writing nested loops.

7-7 REMARKS ON TECHNIQUES, STYLE, AND PITFALLS

7-7-1 Writing More Efficient Programs

When the value of a variable is used in a computation, the computer must fetch the value from the memory location assigned to that variable. When a

new value is assigned to a variable, that value must be stored in the corresponding memory location. Although there might only be a very small difference, it does take longer for the computer to fetch or store the value of a subscripted variable than the value of an unsubscripted variable. The difference lies in the fact that the computer must actually do some computing to find the location of a subscripted variable, while no such computation is required for an unsubscripted variable.

Let us see why this is so. When you refer to an unsubscripted variable, the program that the computer executes actually includes the direct reference to the memory location for that variable. That is, the identification of the name of the variable with the location where the variable is stored has been taken care of by the compiler. Let us be more explicit. The situation is that the memory locations in the computer are internally referred to by *addresses* 1, 2, 3,.... What happens, then, is that the compiler might, say, assign your variable named TOTAL to the memory location with address 559. Then when your Pascal program instructs the computer to store a number in TOTAL, the machine language program that the compiler constructs has a corresponding command instructing the computer to store a number into the memory location with address 559.

Unfortunately, this is not a feasible approach for subscripted variables. This is because subscripts are generally variables or expressions with values that are computed as the program executes. The particular variable referenced depends on the subscript value, so there is no way that the specific addresses of locations can be inserted into the machine language program before it is executed. The approach taken is therefore the following. The computer keeps track of a *base address* for each array. This is the address of the location of the first member of the array, e.g., A[1], or A[1,1], or A[1,1,1]. The members of the array are then stored in memory locations numbered consecutively, beginning with the base address. Any particular member can be located by a simple (but necessarily time-consuming) computation.

The moral of all this discussion is that, if you have a computation in which the value of a subscripted variable is referred to many times, it might be worthwhile to fetch its value once and store it as a temporary variable. For example, the statements

```
X := 3.5 * A[1,2,7] + B/A[1,2,7];
Y := (A[1,2,7] - 8.5) * (A[1,2,7] + 25.2)
```

can be replaced by

```
T := A[1,2,7];
X := 3.5 * T + B/T;
Y := (T - 8.5) * (T + 25.2)
```

In this example, only three computations of the address of the location of A[1,2,7] have been saved. There are other advantages though. The second version is less likely to have a typographical error creep into it. Furthermore,

it's easier to read. You don't have to keep looking back to see whether all those subscripted variables really are the same. An extra storage location is a small price to pay for these advantages.

In Section 6-7 we discussed how you can make your programs more efficient by moving computations that don't need to be repeated *outside* the FOR loop. Keep in mind that subscript usage also involves computation and don't put subscripted variables into loops where they don't belong. Here's an example. Suppose 10 pizza parlors each sell five different sizes of pizza and you want to find the best buy (in terms of price per square inch). You might start by storing the diameters (in inches) of the various sizes in a one-dimensional array called SIZE and the prices in a two-dimensional array PRICE, so that PRICE[I,J] is the price (in cents) of the Ith pizza at the Jth restaurant. Now the only computation is that of the price per inch:

```
PPI := PRICE[I,J] * 4.0/(PI * SIZE[I] * SIZE[I])
```

To generate a table of PPI for comparative purposes, the easy approach is to put this computation into a loop:

```
FOR J := 1 TO 10 DO
  FOR I := 1 TO 5 DO
    BEGIN
      PPI := PRICE[I,J] * 4.0/(PI *
        SIZE[I] * SIZE[I]);
      WRITELN (I,J,PPI)
    END
```

But clearly the computation of 4.0/PI can be done outside both loops. (It should only be done once, not 50 times!) Furthermore, the computation of SIZE[I] * SIZE[I]—which requires two computations of the location of SIZE[I] in addition to the multiplication—should only be done five times, once for each size. A bit of rearrangement (with an addition of two temporary variables) gives the following code, in which the area is computed only once for each size.

```
T := PI/4.0;
FOR I := 1 TO 5 DO
  BEGIN
    AREA := SIZE[I] * SIZE[I] * T;
    FOR J := 1 TO 10 DO
      BEGIN
        PPI := PRICE[I,J] / AREA;
        WRITELN (I,J,PPI)
      END
    END
```


To determine whether a more efficient program segment is really “better,” you should weigh the answers to such questions as: Which version is easier to read and understand? Which is simpler and less prone to error? (An obvious pitfall in the first version here is the possibility of muddling the arithmetic by forgetting the parentheses around the denominator; but in the second you might forget a BEGIN or END.) Moreover, how important is efficiency in this case—will the program be executed only once or a million times?

There’s a space trade-off, too. When only one or two temporary variables are introduced to save a good bit of time, the choice is clear. But what if a lot of storage space is needed to save a small amount of time? The detailed analysis of space time trade-offs can be complicated and is beyond the scope of this book.

7-7-2 Saving Memory Space

We have seen that loops can waste a lot of time unless we write them carefully. Similarly, arrays can waste a lot of space. It is often worthwhile to give a bit of thought to how many memory locations are *really* needed to get the job done. Look again at Example 7-2. There we assumed there was an array named GRADE set up to hold the 200 grades and an array F to hold the grade totals. Altogether we used 300 locations. If we don’t want to store the grades for some purpose, we only need 100 locations to get the job done. If we declare only the array F of 100 locations to hold the totals, the following piece of code will read in the grades and store the totals in F.

```
FOR I := 1 TO 100 DO
    READ (F[I]);
FOR I := 1 TO 100 DO
    BEGIN
        READ (NEXTGRADE);
        F[I] := F[I] + NEXTGRADE
    END
```

What happens here is that the first set of grades is read in and stored in the array F. Then, as the next set is read in, each grade is added to the corresponding grade of the previous set, and the total is then stored back in F. You will find that it is often possible to carry out computations in this way as the data is read in, and hence save on memory space.

7-7-3 Identifying Your Input Data—Guarding Against Mixed-Up Data

So far in this chapter, all of the programs to read and total student grades depend critically on having the cards with the grades on them presorted in the correct order. This is asking a lot—whoever is punching the cards might

skip one or might drop the deck of cards and scramble them up. Clearly we need some kind of identification on the data cards. Suppose each student has a unique identification number between 1 and 100. Then we can type the identification number, followed by the grade, on each data card. The following statement will then read in the first set of 100 grades and store them in array F.

```
FOR I := 1 TO 100 DO  
  READ (J, F[J])
```

Notice that the grade for student J is stored in the Jth spot in the array, no matter what order the grade cards are in. Now if the second set of grade cards is similarly identified, we can read the grades and add them to the total for the Jth student by the following statement.

```
FOR I := 1 TO 100 DO  
  BEGIN  
    READ (J, NEXTGRADE);  
    F[J] := F[J] + NEXTGRADE  
  END
```

What if, on each card, the grade is punched *before* the identification number?

7-8 A PROGRAMMING EXAMPLE

Let us expand our grade computation example into a complete program to compute and print out the semester grades for a set of 100 students identified by numbers 1...100. Assume there are three components to the grade:

1. A mid-term exam grade with a maximum of 100 points
2. A total homework grade with a maximum point total of 150
3. A final exam grade with a maximum of 200 points

To add a bit more complexity, suppose that in computing the total grade, we want to weight these components so that the mid-term and the homework each count 25 percent of the total grade, while the final counts 50 percent of the total grade. We see that we can do this by multiplying both mid-term and final grade by 1.5 as we add them to the total.

How do we go about writing the program? As a first step, anything vague about the given problem specification needs to be made precise. For example: What happens if a student doesn't take one exam? Can the grades for the different components, as well as for the different students, get mixed up (e.g., by dropping the deck of cards)? Are the grades always integers? Precisely what output is required? After answering these questions, we arrive at the following:

Specification of Input Data

Each card contains (1) an integer specifying the identification number of the student; (2) an integer specifying whether this is component 1, 2, or 3 of the grade; and (3) a grade, which need not be an integer. Furthermore, there are exactly 300 data cards, one for each grade component for each student. (A student who misses an exam gets zero.)

Specification of Output Results

A table will be printed (with appropriate column headings) giving student identification number followed by the weighted grade total. The students should be listed in order: 1, 2,...,100.

Once the specification looks reasonably complete, we then need to think about how the computation can be carried out. Since the grade components can be all mixed up, there's no reason to read the cards 100 at a time. We just need a loop to read 300 cards and add the grade on each of them to the proper total. We have seen in the last section how adding a grade to the correct total is readily accomplished by subscripts. One thing we mustn't forget, however: Before beginning to accumulate the total grades by adding in components, the locations where the grades are being stored *must be initialized to zero*.

Now, how do we handle the weighting of the grades? One way is to store the weights in another array; i.e., store 1.5 in WEIGHT[1], 1.0 in WEIGHT[2], and 1.5 in WEIGHT[3]. Then we can use the same trick that we used to add a grade to the correct total: If a grade is a final exam grade (component 3), we multiply it by WEIGHT[3] before adding, and similarly for the other components. If you are an experienced programmer, another point might occur to you. Setting the values of the weights in the program reduces the generality of the program. It is much better to read them in as data. Similarly, it is dangerous to count on there being exactly 100 students. We might use 100 as a *maximum*, to set the array size, and then read in the exact number for any particular computation.

At this point we are ready to draw a flowchart for the overall program. See Figure 7-5. If this were a complicated program, we might proceed by drawing more flowcharts, breaking down each step (each rectangle in a particular flowchart) into finer and finer levels of detail. But for simple programs (which is what this is) more detailed flowcharts are really not necessary. The next steps to be carried out then are:

1. Decide what arrays, variables, and constants are needed. Choose names for them.
2. Write the program.
3. Document the program: i.e., insert comments as necessary to explain data card format, what the program does, etc.

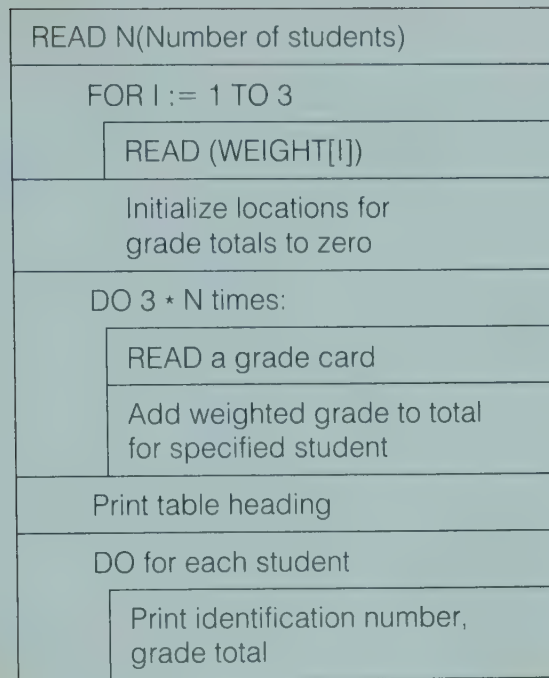


Figure 7-5

Carrying out these steps, we have arrived at the following program.

```

PROGRAM GRADECOMP (INPUT,OUTPUT);
(* PROGRAM TO COMPUTE WEIGHTED TOTAL OF
  3 GRADE COMPONENTS FOR EACH OF N
  STUDENTS (N <= 100).
  INPUT DATA: N
                LIST OF 3 WEIGHTS
                3*N GRADE CARDS IN FORMAT:
  IDENT  COMPONENT  GRADE
  WHERE IDENT IS STUDENT NUMBER (BETWEEN
  1 AND N) AND COMPONENT IS 1, 2, OR 3. *)
VAR IDENT, COMPONENT, I,J,N : INTEGER;
    GRADE : REAL;
    WEIGHT : ARRAY[1..3] OF REAL;
    TOTGRADE : ARRAY[1..100] OF REAL;
BEGIN
  READ (N);
  FOR I := 1 TO 3 DO
    READ (WEIGHT[I]);
  FOR I := 1 TO N DO
    TOTGRADE[I] := 0.0;
  FOR I := 1 TO 3*N DO
    BEGIN
      READ (IDENT,COMPONENT,GRADE);
      TOTGRADE[IDENT] := TOTGRADE[IDENT]
        + WEIGHT[COMPONENT]
        *GRADE
    
```

```

        END;
    WRITELN ( '      GRADE LIST' );
    WRITELN ( ' *****' );
    WRITELN;
    WRITELN ( ' I.D.      GRADE' );
    WRITELN;
    FOR J := 1 TO N DO
        WRITELN ( J:5, TOTGRADE[J]:10:2 )
    END.

```

Once the program is written, our work is not yet done. We need to *test* it by running it and to *debug* it, i.e., find the errors (popularly known as "bugs") and eliminate them. The compiler will catch the syntax errors and tell us about them. Then we need to *make up a set of test data* to try the program on. Now you see one advantage of making N a variable: we can make up a fairly small set of test data, say for $N = 3$. The test data should have the grade cards for different components and for different students well scrambled, so that we can see that the program works even if the grade cards are dropped on the floor and mixed up. Finally, we need to *check the answers* that the computer prints out. There's only one way to do this. You must tediously compute by hand (i.e., by pocket calculator) what the answers should be for the given test data. *Never assume that just because your program runs and gives you answers it is correct!* A complicated program requires extensive testing with different sets of data to check behavior for various special cases.

To test our program GRADECOMP, we ran it with the following data:

3		
1.5	1.0	1.5
1	1	95
2	3	100
1	2	66
1	3	88
3	2	99
3	1	77
2	2	55
2	1	85
3	3	20

As you see, data for the different students and different tests is well scrambled. The output we got was

```

        GRADE LIST
*****

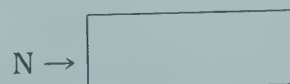
    I.D.      GRADE
    1      340.50
    2      332.50
    3      244.50

```

Happily, the answers are correct!

Suppose that you have designed your program carefully, written out the commands in what you think is correct syntax, and typed your program without any typographical errors. If you have been careful, the compiler will agree that there are no syntax errors and will attempt to execute your program. If you have been *extremely* careful, the program will run on the first try and give correct answers. But what if it doesn't? From time to time in this book we will try to provide you with some ideas as to what to do next. Here are some hints on finding errors in simple programs.

1. *Make sure you have all available information; gather all the clues you can.* This information should include (a) a printout of the program (including messages from the compiler), (b) any output generated, (c) the actual input data, and (d) any messages generated by the system when the program ran. (For example, the program may have stopped upon attempting to read nonexistent data or data of the wrong type. These are important clues.)
2. *Make sure your information is valid.* We, alas, have more than once had a student present us with a copy of yesterday's version of a program, together with last week's data, and ask us to help find out what went wrong with today's run. Sounds ridiculous, doesn't it? But you'll be surprised at how easy it is to think, "Well, I just made a small change that couldn't possibly affect this part of the program where my error seems to be, so...." Don't fall into the trap of making such assumptions! You could waste hours—even days—of valuable time.
3. *Check out all the clues you have.* If the program ran to completion, but just gave bad answers, suspect a data error, an error in an arithmetic expression, or a careless reference to the wrong variable in the wrong place. If the program stopped upon attempting to read data of the wrong type, look first for a data error, then at your READ statement and the declaration of the variable whose value is being read. If you got some output, but not as much as you expected, look to see how far your program seemed to execute correctly. This should narrow down the region of the program in which to search for an error. You will learn how to follow up on other clues as you gain experience.
4. *Go through your program by hand.* If you've followed up on all the computer-generated clues and still can't locate what's wrong, pretend you are a computer and execute your program. Go through the program statement by statement, doing exactly what the computer would do. For example, if the first statement is READ(N), make a box labeled N to stand for the memory location named N.



Then look for the first number in your data and write it in the box. As new values are assigned to variables, change the contents of the boxes.

As you execute loops, make sure you increment the index variable and perform the termination test as you should. Remember that the computer doesn't assume *anything*. If a variable isn't initialized as it should be, its value is not zero (or whatever other handy value you might like it to be) but some "garbage" number. In short, *if* you understand how a computer executes the statements in your program and *if* you are scrupulous in executing the program in just that way, you are bound to eventually discover your error. This is a tedious process, of course. But it is valuable enough that some people recommend hand execution as a step to be carried out *before* running a program. In fact, a sort of mental execution of the program, without writing everything down, is an excellent way to catch errors in your program design.

5. *Generate more clues.* The worst possible situation is when a program seems to execute all right (i.e., there are no computer error messages), but there is no output. Assuming that this isn't for some trivial reason (e.g., you forgot to put any WRITELN statements in the program), you have very little to go on to try to track down your problem. Even if you got some output, the information may be inadequate. The solution is to try to get more information about what happened. An easy way to do this is to insert more WRITELNs into the program. Print out the input data as you read it in. Print out the values of variables as they are altered. Print out the values of loop indexes as loops are executed, to see that they are executed the right number of times. For example, if your program has a loop

```
FOR I := 1 TO 100 DO
  BEGIN
    .
    .
    .
  END
```

you might insert after the BEGIN a statement such as

```
WRITELN ( 'LOOP EXECUTED FOR I = ', I );
```

You may end up with what looks like too many clues—pages of output—if there are long loops in your program. But it will be worth it if you can then follow through what the computer is doing and spot the error. In long programs, it's often a good idea to include extra WRITELN statements as the program is written. This is on the theory that there will almost surely be some bugs and you might as well start right off with extra clues. The extra WRITELNs should be inserted so they can be readily removed from the final program. For example, you might write them so they can be easily spotted, as

```
( * DEBUG * ) WRITELN( ... )
```

6. Learn about your system's debugging tools. Your computer system may provide you with other information than simply a flat statement of why your program stopped. For example, it probably will tell you what statement it was executing when it stopped. It may tell you the values of your variables at the time of termination. Sometimes the error messages are very cryptic. Don't guess what they mean; find out! There's another possibility. There may be simple ways to tell the computer to print out values of important variables as the program executes, without inserting a lot of extra `WRITE` statements in your program. Facilities you can invoke to monitor the execution of your program in this way are sometimes called *trace routines*. There should be a manual available to you that describes your local system's implementation of Pascal and the debugging facilities it provides. Learn to consult this manual when you have problems.

7-10 EXERCISES

- 7-1. What changes need to be made in the example of Section 7-8 to generalize it to the case where the number of grade components, instead of being fixed at three, can be read in by the user?
- 7-2. Some Pascal compilers allow you to set all elements of an array to zero (or any other constant) by writing

$$X := 0.0$$

where X is the array name. Determine whether your compiler allows this.

- 7-3. Given the following declarations:

```
VAR A : ARRAY[1..10] OF INTEGER;
    B : ARRAY[1..10, 1..10] OF REAL;
```

- (a) Write a program segment to replace each value in A by 3 times its current value and each value in B by 6 times its value.
- (b) Write a program segment to sum the values in each row and column of B. Print each row sum and each column sum. (Recall that by convention the first subscript of a two-dimensional array numbers the row and the second subscript numbers the column.)
- 7-4. What is printed by the following FOR statements?

```
(a) FOR I := 1 TO 3 DO
    FOR J := 1 TO 4 DO
        WRITE (I, J)
    (b) FOR I := 1 TO 3 DO
        BEGIN
            L := I + 1;
            FOR J := L TO 3 DO
                WRITE (I, J)
            END
```

- 7-5. If an assignment statement takes, on the average, 0.01 seconds to execute, how long will the following program segment take? (Neglect any overhead in the FOR loops and express the answer in any convenient units of time.)

```

FOR I := 1 TO 5 DO
  FOR J := 1 TO 73 DO
    FOR K := 1 TO 24 DO
      FOR L := -3 TO 56 DO
        FOR M := 1 TO 60 DO
          FOR N := -10 TO 89 DO
            FOR X := 3.14159

```

- 7-6. Given the following declaration

```
VAR A : ARRAY[1..30] OF INTEGER;
```

- Write a program segment to "circular shift" the elements of A to the right by distance 1. (This means that A[I] should be replaced by A[I-1] for 2 ≤ I ≤ 30 and A[1] should be replaced by A[30].) Hint: Remember the FOR loop with DOWNTO.
 - Write a program segment to "circular shift" the elements of A to the right by a distance D. D, an integer between 1 and 10 (inclusive), is to be read from a data card. (In this case A[I] is replaced by A[I-D] for D+1 ≤ I ≤ 30 and A[1], A[2], ..., A[D] are replaced by A[31-D], A[31-D+1], ..., A[30], respectively.)
- 7-7. Turn the pizza example of Section 7-7 into a complete program to read in pizza prices and output a table of prices per square inch. Follow the example of Section 7-8 to develop a problem specification including possible generalization, precise input-output formats, etc.
- 7-8. A consumer group has been alarmed about the recent rapid rise in price of a certain grocery item. The group sampled grocery store prices of that item on a monthly basis throughout 1980, 1981, and 1982 and observed that the price varied from 84 cents to 155 cents as the following table shows:

Month

Year	1	2	3	4	5	6	7	8	9	10	11	12
1980	87	89	89	89	85	84	90	100	90	90	90	100
1981	95	95	95	90	85	100	110	120	115	100	110	97
1982	120	125	130	135	140	145	150	155	155	145	147	140

These 36 prices are now available on data cards, and a program is needed to analyze the price change. Write a Pascal program that reads in this data and then carries out the following analyses, printing out the results, appropriately identified.

- (a) Compute the percent increases in the monthly prices according to the formula:

$$\% \text{ increase from month } i-1 \text{ to month } i = 100 * \left(\frac{\text{month } i \text{ price}}{\text{month } i-1 \text{ price}} - 1 \right)$$

- (b) Compute the average price for each year.
 - (c) Compute the percent increase of the yearly average prices using a formula similar to that given in (a).
- 7-9.** The transpose of an $N \times N$ matrix A is defined to be another $N \times N$ matrix B such that $B[I,J] = A[J,I]$ for all I,J . Write a program that:
- (a) Reads an integer N ($N \leq 10$)
 - (b) Reads an $N \times N$ matrix row by row and stores it as A
 - (c) Transposes A *in place*; i.e., A is overwritten by its transpose as the transposition takes place
 - (d) Prints the transposed A

SUMMARY OF CHAPTER 7

Arrays of subscripted variables must be declared like any variables. The lower and upper bounds of the subscripts must be integer constants. A subscripted variable with a subscript whose value is outside the lower and upper bounds will cause an error.

Arrays are naturally processed by FOR loops in which the loop index corresponds to the array subscript. Arrays with more than one subscript are usually processed by nested FOR loops, in which one FOR loop (the “inner” loop) is contained within the code executed under the control of another (the “outer” loop). In each pass of the outer FOR loop, the statements within the inner FOR loop will be executed repeatedly as specified by the lower and upper bounds on the inner loop. Deeper nesting of FOR loops is also legal.



LOOP CONTROL WITH WHILE OR REPEAT STATEMENTS

8-1 INTRODUCTION

Suppose we have a robot in a shoe factory that can sew on the soles of shoes for us. On some occasions, we might wish to give the robot an instruction such as "Sew on the soles of 300 pairs of shoes." In this case, the robot will sew on the soles of exactly 300 pairs of shoes and then will either stop or go on to perform other tasks. On other occasions, we might wish to give the robot instructions such as "Sew on soles as long as the supply of soles lasts" or "Sew on soles as long as the light is on" or "Sew on soles until the light is turned off." In these cases, the robot might sew on many pairs of soles or might sew on only a few, depending on how long the supply of soles lasts or how long the light is left on. This illustrates three different ways in which the repetition of a task performed by a robot is controlled. First, a task may be repeated a prespecified number of times, such as 300. Second, a task may be repeated as long as a certain condition remains true (e.g., "as long as the supply of soles lasts" or "as long as the light is on"). Finally, a task may be repeated *until* a certain condition *becomes* true ("until the light is turned off"). In a similar fashion, there are three ways to instruct a computer to carry out a computational task repeatedly. In Chapter 6, we saw that we can use the FOR statement to repeat a task a fixed number of times as specified by the initial and final values of the index or control variable. In this chapter, we shall see how we can instruct the computer to repeat a task for as long as a given condition is true or until a given condition becomes true.

8-2 THE WHILE STATEMENT

We might formally instruct the robot to sew on the soles by a statement such as

```
WHILE LIGHT IS ON DO  
    SEW ON ONE PAIR OF SOLES
```

In this instruction there is a *condition*: LIGHT IS ON. If the condition is true (i.e., the light is on), the robot will carry out the specified *action* of sewing on one pair of soles. After that, the robot will check whether the light is still on. If the condition is still true, the robot will sew on another pair of soles, and then again check whether the light is on. As soon as the robot discovers that the condition LIGHT IS ON is false (i.e., the light is off), it will no longer sew on soles but go on to other tasks.

Pascal has a WHILE statement exactly like this. Let us give an example. Suppose we have a deck of cards on which the ages of the members of a club are punched. We want to read the cards one at a time and pick out the first age larger than or equal to 65. The following statements will do the job.

EXAMPLE 8-1

```
AGE := 0;  
WHILE AGE < 65 DO  
    READ (AGE);  
    WRITELN ('THE FIRST ONE ENCOUNTERED IS', AGE)
```

Note that the WHILE statement contains a condition, $AGE < 65$, and an action, READ (AGE). The symbol $<$ means "less than." Hence the condition we are checking is whether the value of AGE is less than 65. The first time around, since AGE was set initially to 0, the condition is satisfied. Consequently, the action READ (AGE) will be carried out. Suppose that the value read is 48. The condition $AGE < 65$ is tested again (with the value of AGE now equal to 48). Since the condition is still true, the READ statement will be executed, another value of AGE will be read in, and the condition $AGE < 65$ will be tested again. Eventually, after reading in a value of AGE that is larger than or equal to 65, the condition $AGE < 65$ becomes false. At this point, the action READ (AGE) will be skipped and the following statement, WRITELN, will be executed. Let us emphasize that nowhere is it specified how many times the action is to be repeated. Indeed, the number of times that the action will be repeated depends entirely on the data to be read in.

This example illustrates the general structure of the WHILE statement. The WHILE statement includes both a condition and an action or task to be carried out. The action is specified by a statement. Just as in FOR statements, complicated tasks may require compound statements for their specification. Execution of a WHILE statement proceeds as follows: If the condition is true, the action will be carried out. The condition will then be tested again. If the condition is still true, the action will be repeated. And so forth. Eventually, when the test shows the condition false, the action will not be executed and the computer will go on to execute the next statement in sequential order after the WHILE statement. Notice that if the condition is initially false, the specified action is not carried out even once.

We shall explain in the next sections the general form of a *condition*. In order to do this, we first need to discuss a new type of variable and how one operates on it.

8-3 BOOLEAN VARIABLES

We have already discussed two types of variables—REAL and INTEGER. Here we introduce another type, BOOLEAN. There are only two BOOLEAN constants, TRUE and FALSE. Hence a BOOLEAN variable can have as its value either the constant TRUE or the constant FALSE. Just as we can declare variables to be REAL or INTEGER, we can declare variables to be BOOLEAN:

```
VAR LIGHT, WARM, READY: BOOLEAN;
```

And just as we can assign numbers to REAL or INTEGER variables, we can assign the values TRUE or FALSE to BOOLEAN variables by means of assignment statements. For example:

```
LIGHT := TRUE;  
WARM  := FALSE;  
READY := LIGHT
```

Here the value of the BOOLEAN variable LIGHT is set to TRUE, the value of the BOOLEAN variable WARM is set to FALSE, and then the value of the BOOLEAN variable READY is set to TRUE by assigning it the value of LIGHT.

Then if we want to print the value of READY, we write

```
WRITELN (READY)
```

and what will be written is

```
TRUE
```

Just as for integers, the default fieldwidth is 10. Thus

```
WRITELN (READY)
```

will print six blanks before TRUE. As usual, we can specify a fieldwidth to change the number of blanks printed. Thus

```
WRITELN (READY : 8)
```

would print four blanks before TRUE.

One way in which BOOLEAN variables and values are very different from REAL or INTEGER variables is that you cannot READ them. Thus if you write

```
READ (READY)
```

you will get a type-error message.

Arrays of BOOLEAN variables can also be defined. Thus we might declare

```
VAR ANSWERS: ARRAY [1..20] OF BOOLEAN;
```

to set up an array of 20 locations in which we can store a set of answers to a true-false test. (But since we can't read in boolean values, this may not be a very practical approach to handling true-false answers!)

BOOLEAN variables are used mostly to represent conditions. For example, the variable LIGHT may represent the informally stated condition "light is on," in the sense that if the value of LIGHT is TRUE then the light is on, while if the value of LIGHT is FALSE, the light is off. We can then replace the informal syntax

WHILE LIGHT IS ON DO ...

by the correct Pascal syntax

WHILE LIGHT DO ...

Conditions may also involve combinations of BOOLEAN variables, as we shall now explain.

8-4 BOOLEAN EXPRESSIONS

Just as we can build up arithmetic expressions by combining the operators +, -, *, and / with REAL and INTEGER constants and variables, we can combine boolean operators with BOOLEAN constants and variables to build up boolean expressions. There are three boolean operators: AND, OR, and NOT. The operators AND and OR are binary operators; i.e., they are used in conjunction with two operands (like the arithmetic operators +, -, * and /). NOT, on the other hand, is a unary operator, taking a single operand (much like the negative sign). Since there are only two values, TRUE and FALSE, that operands and results can take on, the effects of boolean operators are commonly defined by simple tables. (See Tables 8-1 and 8-2.)

Table 8-1
BOOLEAN OPERATORS AND, OR

Values of X Y		Value of X OR Y	Value of X AND Y
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE

Table 8-2
BOOLEAN OPERATOR NOT

Value of X	Value of NOT X
TRUE	FALSE
FALSE	TRUE

These are not hard to remember: they correspond, in fact, exactly to what one always means by the words “and,” “or,” and “not.” As we pointed out in the last section, BOOLEAN variables can represent conditions. Thus X might mean LIGHT IS ON and Y might mean ROBOT IS HAPPY. Then X OR Y represents the condition LIGHT IS ON OR ROBOT IS HAPPY. This condition is true whenever *either* the light is on *or* the robot is happy, i.e., whenever *either* (or both) of the subconditions is true. Looking at the table, you will see that X OR Y is FALSE if and only if *both* X and Y are FALSE. Similarly, for X AND Y to be TRUE, *both* X and Y *must be* TRUE. Finally, NOT X (in our example) would mean “the light is *not* on”—which is FALSE when the light is on and TRUE when the light is off. That is, the NOT operator forms the **negative** of the condition it is applied to.

Notice that we are combining simple conditions, represented by single variables, into more complicated conditions by using boolean operators to form boolean expressions. *In Pascal a condition is defined to be any expression that evaluates to either TRUE or FALSE.* Thus, when we write WHILE statements, the “conditions” may be BOOLEAN variables or more general boolean expressions. Since they always evaluate to either TRUE or FALSE, they tell the computer whether or not to execute the task specified in the WHILE statement.

Just as arithmetic expressions may involve long, complicated combinations of operators and variables, boolean expressions can also be complicated. Thus we might write

$$(X \text{ AND } Y) \text{ OR } (Z \text{ AND } (\text{NOT } X)).$$

Let us see how this evaluates when X is FALSE, Y is TRUE, and Z is TRUE. First, (X AND Y) is FALSE, since X is FALSE. Next, (NOT X) is TRUE, since X is FALSE. Then, since both Z and (NOT X) are TRUE, (Z AND (NOT X)) is TRUE. Finally, the whole expression, being an OR of two operands of which one is TRUE, is TRUE.

Also as for arithmetic expressions, we can avoid using so many parentheses by adopting some rules of precedence. The order in which the boolean operators are applied is (1) NOT, (2) AND, and (3) OR.

Thus the example above can be equally well written as

$$X \text{ AND } Y \text{ OR } Z \text{ AND NOT } X$$

But if this looks confusing to you, put in the parentheses.

8-5 RELATIONAL OPERATORS AND CONDITIONS

In writing computer programs, we are unlikely to want to use conditions such as LIGHT IS ON or ROBOT IS HAPPY. A more likely sort of condition is AGE = 65, as we used in Example 8-1. Indeed, many of the conditions we

shall use involve comparisons between REAL or INTEGER values. Thus we might have conditions like

```

      HEIGHT = 72.0
    EXAMGRADE > 75
  (WEIGHT/HEIGHT) < 2.0
      WEIGHT < (2.0 * HEIGHT)

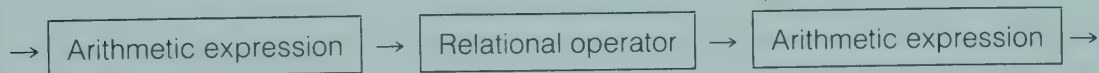
```

What we are doing here is forming expressions which in turn state relationships between REAL or INTEGER expressions. If the relationship holds, the expression is true; otherwise, the expression is false. Thus `HEIGHT = 72.0` is true if the value of the variable `HEIGHT = 72.0`. When we say “`HEIGHT = 72.0` is true,” we mean that the expression `HEIGHT = 72.0` actually evaluates to the BOOLEAN constant `TRUE`. For example, if `X` is defined as a BOOLEAN variable and we write

```
X := HEIGHT = 72.0
```

the expression on the right side is evaluated as `TRUE` and `X` is then assigned the value `TRUE`.

Notice that we can in general use arithmetic expressions—such as `(WEIGHT/HEIGHT)` or `(2.0 * HEIGHT)`—in writing conditions. The general form of these conditions, which are sometimes called *relational conditions*, is:



We have seen that the relational operators include *equals* (`=`), *less than* (`<`), and *greater than* (`>`). The complete list of relational operators available for forming conditions involving arithmetic expressions is given in Table 8-3. Note that the last three operators are written as *two characters without any space between them*.

We have seen before that we must be careful when we use different types of numbers. What happens when we combine REALs and INTEGERs with

Table 8-3
RELATIONAL OPERATORS

Symbol used in Pascal	Meaning
<code>=</code>	Equals
<code>></code>	Greater than
<code><</code>	Less than
<code><></code>	Not equal
<code><=</code>	Less than or equal
<code>>=</code>	Greater than or equal

relational operators? For example, what if EXAMGRADE is an INTEGER variable—is the condition $\text{EXAMGRADE} > 75.5$ legal? The answer is that there is no problem in comparing REALs with INTEGERS. Just as in an arithmetic expression, the INTEGER value is converted to a REAL, so the comparison is straightforward. Suppose, for example, that the value of EXAMGRADE is 80. The INTEGER 80 is converted to 80.0 and then the condition

$$80.0 > 75.5$$

is evaluated to TRUE.

Since relational conditions evaluate to TRUE or FALSE, they may be combined into boolean expressions in the same way as BOOLEAN variables. Thus we may write

$$(\text{AGE} < 65) \text{ AND } (\text{AGE} > 18)$$

which evaluates to TRUE when the value of AGE is *both* less than 65 *and* greater than 18. In this example, notice the parentheses around the relational conditions. These are required in Pascal. This is because relational operators can be used to compare boolean expressions as well as arithmetic expressions. (By convention, FALSE is “less than” TRUE.) When the computer evaluates complicated expressions involving relational, boolean, and arithmetic operators, the NOT and the unary minus are applied first; then the AND has the same precedence as and and $*$; the OR has the same precedence as $+$ and $-$; and the relational operators are of lower precedence than any of these. Thus in the example above AND has higher precedence than the relational operator $<$, and, if we remove the parentheses, the condition

$$\text{AGE} < 65 \text{ AND AGE} > 18$$

will be interpreted as

$$(\text{AGE} < (65 \text{ AND AGE})) > 18$$

The fact that the data types are all wrong for this expression to make sense should, of course, cause the compiler to generate error messages. But you need to know that the messages are caused by the way the compiler interprets expressions. Here's a more complicated example for you to ponder. The expression

$$(-X * 5 > Y) \text{ OR NOT } (X = -4.5) \text{ AND } (A \text{ OR } B)$$

is equivalent to

$$(((-X) * 5) > Y) \text{ OR } ((\text{NOT } (X = -4.5)) \text{ AND } (A \text{ OR } B))$$

Assuming that $A = \text{TRUE}$ and $B = \text{FALSE}$, for what INTEGER values of X and Y will this expression be TRUE?

8-6 READING UNKNOWN AMOUNTS OF DATA

In preceding chapters, we explained how lists of data items can be read into the computer by a FOR loop. To do this, however, the upper bound on the loop index (that is, the number of items in the list) must be specified. This is usually done by first reading in an integer that gives the count of the number of items in the list. But counting the items is not only a nuisance but also must be done with no mistakes. If you have one less data item than is specified, your program will abort upon attempting to read nonexistent data. If you have one more than is specified, the last piece of data will never be used. Furthermore, it is often inconvenient to have a program that can only accept a fixed amount of data. Look back at the example of Section 7-8. There we had to assume that grade cards with grades of zero were prepared for students who didn't take an exam. Otherwise we couldn't have assumed that our program should read in exactly $3 \cdot N$ cards, where N is the number of students. It would be much handier just to prepare cards for students who took the exams and then process them by something like:

```

WHILE THERE IS MORE DATA DO
  BEGIN
    READ GRADE CARD
    PROCESS GRADE
  END

```

An easy way to do this kind of thing in Pascal is to put some sort of meaningless value (e.g., a negative number if all data are positive) at the end of the data and then stop reading after the meaningless value is read. We must then, of course, be careful not to compute with the last value read. This means that we must read a value, then test, then process the data (if the number does not indicate the end of the data), then read again, etc. In the example where we read and process student grades (Section 7-8), we might put zero for the identification number to signal the end of the list of grades. That is, after the last valid grade card we type one that reads:

0 0 0

Then, in outline, the processing loop would look something like:

```

READ ( IDENT, COMPONENT, GRADE );
WHILE IDENT > 0
  BEGIN

```


Process grade card

```
      READ ( IDENT , COMPONENT , GRADE )  
      END
```

This says to read and process another card as long as we have not yet reached the end of the data, which will be indicated by the 0 value of IDENT.

To provide you with another example of the use of a WHILE to terminate data processing, let us look more closely at Example 8-1. Suppose there is no guarantee that there are any members as old as 65. We would want to make sure that the computer, if it doesn't find any values of AGE which make the $AGE < 65$ condition FALSE, doesn't attempt to read nonexistent data. We can do this by putting a zero at the end of the data and testing two relational conditions:

```
      WHILE ( AGE < 65 ) AND ( AGE > 0 ) DO  
      READ ( AGE )
```

This will cause the computer to continue to read values of AGE as long as *both* $AGE < 65$ and $AGE > 0$. So if there are no values of AGE that are greater than 65, the reading will stop after the value of 0 is read.

As soon as we write a loop that can stop because of either of two conditions, we notice another problem. Depending on which condition caused the reading to stop, we would want to print out different messages. In the preceding example, we would want the computer to print either the first age encountered that is greater than or equal to 65 or a message such as "No member is 65 or older." In general, depending on which of several conditions caused a loop to terminate, we would want the computer to take different actions. In Chapter 9 we will explain the Pascal statement that can be used to accomplish this.

As a last example in this section, let us show you how WHILE loops are used to terminate interactive programs. If you have been working at a terminal, you are familiar with what we are talking about. When you are finished using the editor, you type a special command, perhaps QUIT, to indicate that fact. The structure of the heart of the editor is then something like that shown in Figure 8-1.

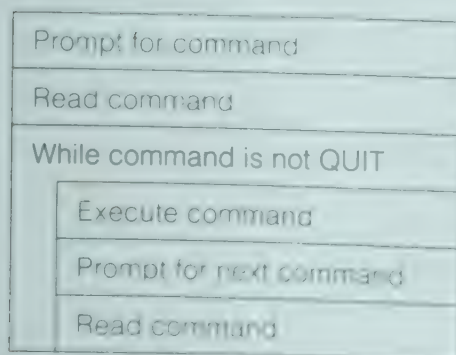


Figure 8.1

Similarly, when you have run your job and are finished using the computer, you indicate that fact to the program you have been interacting with by typing something like LOGOFF or BYE. That program also is structured as a loop to accept and execute commands until you tell it you want to stop. These programs are very much like our robot who continues to make shoes until somebody turns the light off. The robot is getting interactive input by way of the light switch.

As a simpler example than editors or robots, we will write an interactive program that you can use to decide where to buy the cheapest pizza. In Chapter 7, pizza prices were compared by reading in an array of prices and diameters and printing a table of prices per square inch. Here we will assume that you want to interactively request the price per square inch for selected pizzas. And you want the computer to ask you whether you are finished. We need some way to express your request to quit in a WHILE loop condition. One way is to set up a variable (let's call it FINISH) that you type in a special value for (let's make it 2) when you are done. We then structure our program as shown in Figure 8-2.

Here's the program that we wrote for our system. As we said before, systems differ widely in how interactive programming is handled. As an exercise, convert this program to your system and try it out.

EXAMPLE 8-2

```
PROGRAM PIZZA ( INPUT / , OUTPUT );
CONST PI = 3.14159;
VAR DIAM, PRICE, PPSI : REAL;
    FINISH : INTEGER;
```

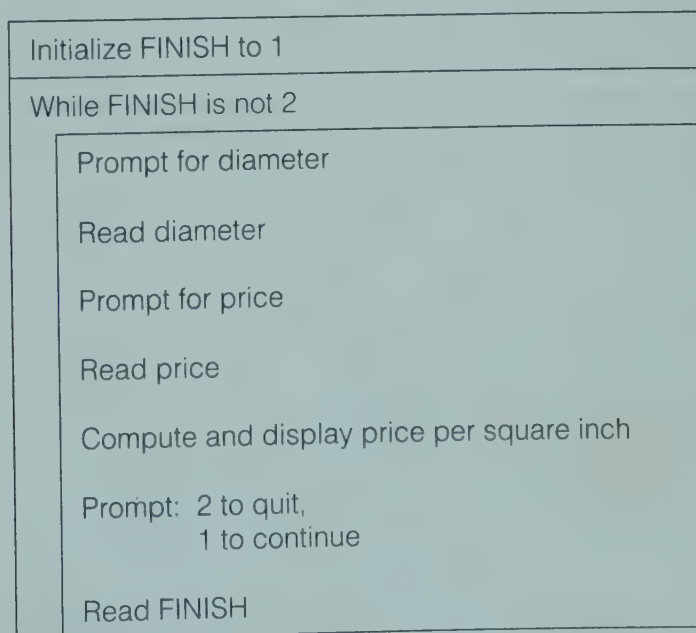


Figure 8.2

```
BEGIN
  Writeln ('TO COMPUTE THE PRICE PER SQUARE INCH'
  FINISH := 1;
  WHILE FINISH <> 2 DO
    BEGIN
      Writeln ('ENTER DIAMETER');
      Readln;
      Read (DIAM);
      Writeln ('ENTER PRICE');
      Readln;
      Read (PRICE);
      PPSI := 4.0 * PRICE / (DIAM * DIAM * PI);
      Writeln ('PRICE PER SQ IN IS', PPSI:6:2);
      Writeln ('DO YOU WANT TO QUIT?');
      Writeln ('ENTER 2 TO QUIT, 1 TO GO ON');
      Readln;
      Read (FINISH)
    END;
  Writeln ('GOODBYE')
END.
```

8-7 LOOPS WITH REPEAT...UNTIL

Recall that in Section 8-1 we said that there was still another way to control the repetition of tasks. This was to specify that a task be *repeated until* a certain event happens, i.e., until a certain condition becomes true. For example, we might tell you to *repeat* the reading of this chapter *until* you understand it thoroughly. In Pascal, there is a way to control repetition in those very same words. Here's an example. To read and total a list of grades, we write the following program. (Assume the list ends with a negative number.)

EXAMPLE 8-3

```
PROGRAM ADD (INPUT, OUTPUT);
VAR TOTAL, NUMBER: INTEGER;
BEGIN
  TOTAL := 0;
  READ (NUMBER);
  REPEAT
    TOTAL := TOTAL + NUMBER;
    READ (NUMBER)
  UNTIL NUMBER < 0;
  Writeln (TOTAL)
END.
```


What happens is that the statements between the REPEAT and the UNTIL are executed until the condition following the reserved word UNTIL is satisfied.

Thus this program executes:

```

READ NUMBER
Add to TOTAL
READ NUMBER
Test condition (number < 0)
Add to TOTAL
READ NUMBER
Test condition (number < 0)
.
.
.
```

Notice that the condition is not tested until *after* the actions specified are carried out. Thus the task specified in a REPEAT loop is carried out *at least once*. This is in distinction to the WHILE loop, where the task is never carried out if the condition is initially false. This is also different from the behavior of FOR loops, where, as we discussed in Chapter 6, the end test is carried out *before* each task repetition. When we use REPEAT...UNTIL to read in data we must therefore be careful that there is at least one data item to be read in addition to the meaningless value denoting the end of the data.

There is another unique feature of the structure of a REPEAT loop. Notice that there is a list of statements between REPEAT and UNTIL, and *not* a single compound statement as is the case with FOR or WHILE. This is because the UNTIL serves to show where the statements to be repeated end, so we don't need to bracket them with BEGIN, END.

Like FOR loops, REPEAT loops and WHILE loops may be nested. Indeed, there is nothing to stop us from nesting different kinds of loops within each other. After all, any kind of statement—including FOR, WHILE, or REPEAT—may be used in specifying the task to be repeated in a loop. Here is an example of a WHILE loop nested inside a REPEAT loop. This program reads a sequence of positive integers and prints their digits, one digit per line. We assume that there is at least one integer in the sequence and that the list is terminated by -1.

EXAMPLE 8-4

```

PROGRAM DIGITS (INPUT,OUTPUT);
VAR NUMBER,D: INTEGER;
BEGIN
  REPEAT
    READ (NUMBER);
```

```

        WHILE NUMBER > 0 DO
            BEGIN
                D := NUMBER MOD 10;
                WRITELN(D);
                NUMBER := NUMBER DIV 10
            END
        UNTIL NUMBER < 0
    END .

```

Study carefully how this program works. The outer (REPEAT) loop simply reads numbers until the end of the data. For each number read, the inner (WHILE) loop, which prints the digits of this number one per line beginning with the least significant, is executed. After all the digits of one number have been printed, the computer leaves the inner loop and goes back to execute the next READ statement—unless the end of the data has been reached.

To emphasize the fact that the termination test is carried out *after* the task is executed, REPEAT loops sometimes are pictured in flowcharts as shown in Figure 8-3.

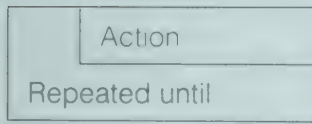


Figure 8.3

Figure 8-4 is the flowchart for Example 8-4.

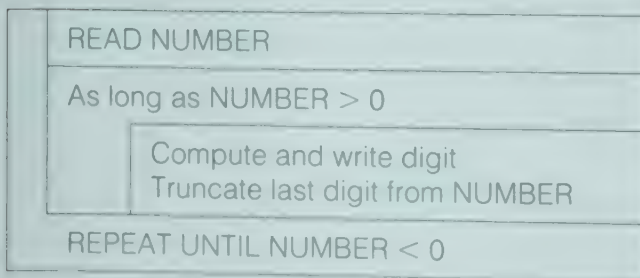
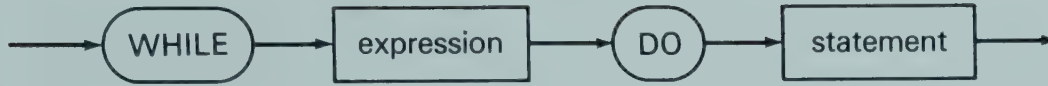


Figure 8.4

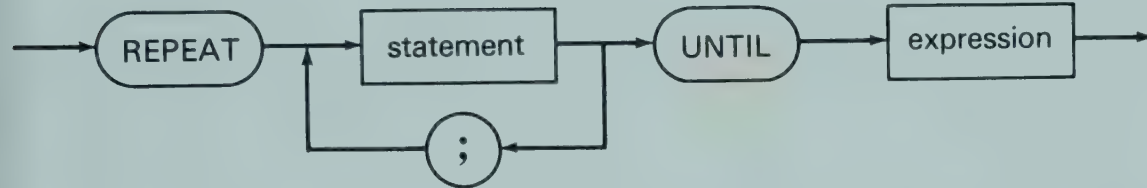
8-8 SYNTAX DIAGRAMS

We have seen that the “conditions” that appear in WHILE and REPEAT statements are formally *expressions* that evaluate to TRUE or FALSE. Here are the diagrams for those statements in terms of expressions.

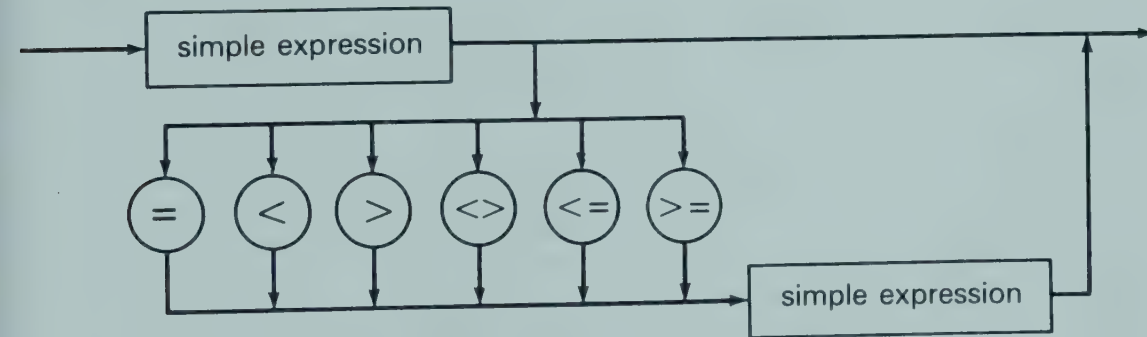
WHILE Statement



REPEAT Statement

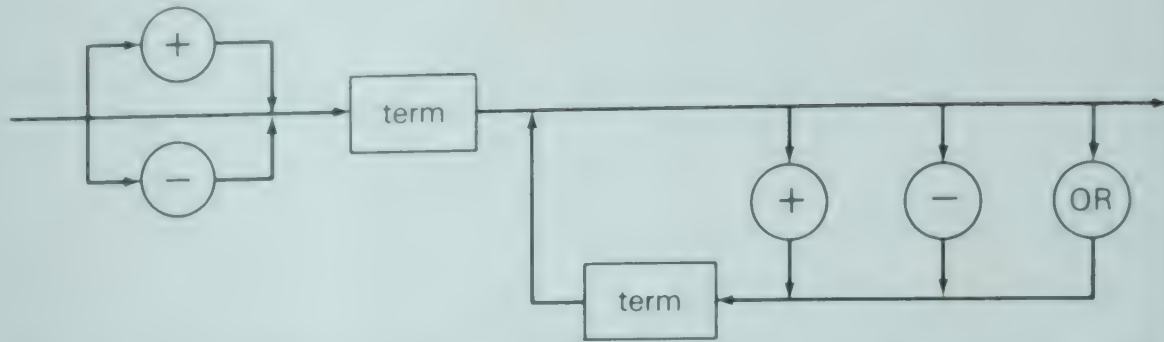


Notice that there is nothing in the Pascal language syntax that requires the expressions appearing in these statements to evaluate to TRUE or FALSE. In fact, the Pascal language syntax does not really distinguish between arithmetic expressions and boolean expressions, or even between boolean operators and arithmetic operators, in constructing those expressions. These distinctions are not in the *syntax* of the language but in its *meaning*, or *semantics*. That is, certain data types may be illegal in certain contexts (you will get a *type-error* message), but the formal syntax, which doesn't distinguish among data types, will not indicate this. Now that we have explained that not all expressions are valid in a WHILE or REPEAT statement (and make sense as conditions), we can proceed to present the general Pascal syntax diagrams that define *expression*. *Expression* is defined by



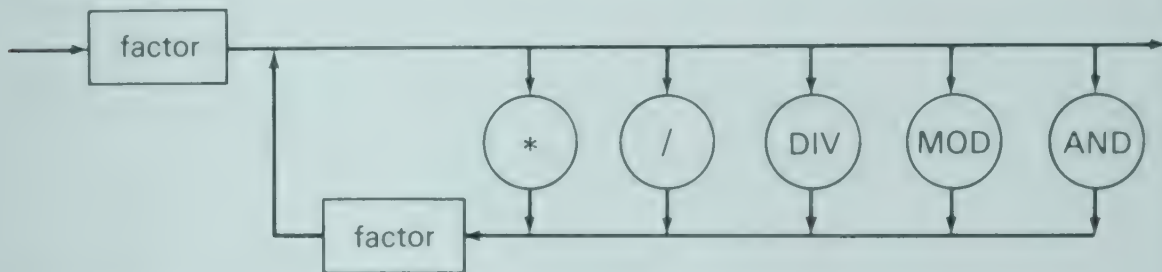
Thus an expression is either a “simple expression”—which we shall see is either an arithmetic or boolean expression—or two simple expressions connected by a relational operator, i.e., what we have previously called a *relational condition*. The syntax diagram for *simple expression* looks much like the one we gave previously for arithmetic expression—but now the boolean operator OR is included:

Simple expression



Now we must expand the definition of *term* by including the boolean AND:

Term



The boolean operations also add one more possibility for *factor*:



We leave it as an exercise for the reader to consolidate this possibility with the syntax diagram for *factor* as given in Section 3-5. (You can check your answer with the complete Pascal syntax diagrams in Appendix B.)

Notice how the rules of precedence for boolean operators are contained in these diagrams. First we have the possibility of forming a factor by applying a NOT to another factor—which must, of course, yield a BOOLEAN value for the NOT to make sense. Next we evaluate terms formed from the multiplication or division operators—or the boolean AND. Then simple expressions are evaluated by addition, subtraction, or the OR operator applied to terms. (You remember, of course, that operations of equal precedence are carried out from *left to right*.) Finally, an expression can be either a simple expression or a comparison of simple expressions that uses one of the relational operators. In the latter case, of course, only a BOOLEAN value can result. What we have been calling *arithmetic expressions* are just simple expressions that yield REAL or INTEGER values.

As a final note, we emphasize that not all expressions constructed according to these formal syntax rules are valid in Pascal. Boolean operators can only be used with boolean operands, and similarly for arithmetic operators. Thus, for example, (NOT 22) and (7 OR 12) are syntactically correct but not valid. Writing such expressions will cause a type error message, since the operands are of the wrong type for the operator. Similarly (7 < 12) * (A > B)

is an attempt to sum two boolean-valued expressions and will cause a type-error message.

8-9 SOME HINTS ON PROGRAMMING STYLE AND PITFALLS

8-9-1 Program Testing

Example 8-4 serves as a handy excuse for us to elaborate on the idea of program testing. In writing this program, there are a number of things one might muddle up through not quite understanding how the loops or the loop conditions work. For example, one might be confused as to when the computer enters and leaves the inner loop. Or the test for the end of the data might not be correct. It is always a good idea to test a program with several sets of data—especially a program involving loop nesting.

Throughout the years, programmers have developed a number of insights into how sets of test data should be chosen. Indeed, the systematic generation of test data sets is a topic of current research. Discussion of this topic in depth is beyond the scope of this text. We will limit ourselves to providing you with the following list of useful hints.

1. Use a large enough set of test data so the computation is not trivial.
2. Test the program with no data to see if that special case is handled correctly.
3. Test data should include tests of all special cases; e.g., when a computation includes a division, look for valid data values that, unless properly handled, cause division by zero.
4. Test extreme values of the data; e.g., if values of the input variable X can be between 0 and 10 (inclusive), test the program for $X = 0$ and $X = 10$.
5. Test the program with “random” data values, i.e., values which have no discernible pattern.
6. If comparative sizes of data values are important, test with values that are equal as well as ones that are different.
7. If data values can be negative or zero as well as positive, include negative, positive, and zero values in the tests.

Let us see how we might apply these hints to testing Example 8-4. This program has been written with the assumption that the numbers to be read in are all positive, hence we can ignore hint 7 above, and hint 4 as well. Hint 2 should worry us. What will happen when the data consists only of the terminating symbol -1 ? It appears that the program will not print anything. This may be an acceptable output for a textbook example, but in some other context we might well want to adjust our program to print a warning message when this happens.

For special cases (hint 3), we would want to use at least three different integers to test this example:

1. One that has only one digit

2. One that has more than one digit
3. One that contains the digit 0

You may believe that all this testing is unnecessary—that programming is easy and you understand it thoroughly. But all programmers make mistakes. The bad programmers are those who don't bother to find their mistakes!

8-9-2 Choice of Loop-Control Statement

Now that you have learned three different ways to write loops, an obvious question is: Which is the *best* way in a given situation? Sometimes the choice is obvious. If you want to repeat a computation exactly so many times, you will probably want to use a FOR loop. However, even then you really have a choice. Recall Example 6-5, which printed a table of I , I^2 , and I^4 for I from 1 to 10. Here we knew that we wanted to repeat the computation 10 times, so we wrote a FOR loop. But we could also use a WHILE, as follows:

EXAMPLE 8-5

```

PROGRAM IFOURTH (OUTPUT);
VAR I, J, K : INTEGER;
BEGIN
    I := 1;
    WHILE I < 11 DO
        BEGIN
            J := I * I;
            K := J * J;
            WRITELN (I, J, K);
            I := I + 1
        END
    END.

```

Or we could equally well use a REPEAT:

EXAMPLE 8-6

```

PROGRAM IFOURTH (OUTPUT);
VAR I, J, K : INTEGER;
BEGIN
    I := 1;
    REPEAT
        J := I * I;
        K := J * J;
        WRITELN (I, J, K);
        I := I + 1
    UNTIL I = 11
END.

```


Notice that the WHILE or REPEAT takes care of the end test, but we have to initialize and increment the counter explicitly. Clearly the FOR statement, which takes care of the initializing and incrementing automatically, is preferable in this case.

The choice between WHILE and REPEAT is not so clear. These statements both control repetition on the basis of whether a condition is true. Sometimes you cannot use REPEAT because of the fact that the task specified in a REPEAT is executed at least one time. If this is no problem, you should use whichever statement seems more natural to you in the given context.

8-9-3 The Problem of Infinite Loops

Both WHILE and REPEAT have a rather nightmarelike pitfall associated with them. This is the possibility of an *infinite loop*, a possibility that we mentioned in Chapter 6 as arising from illegally changing the index inside a FOR loop. You should recall that an infinite loop is just what the name implies—a task that is repeated over and over forever. This should not be an unfamiliar idea to you. Remember the story of the sorcerer's apprentice, who set a broom to work fetching water but forgot how to tell it to stop. Like the broom, the computer will mindlessly repeat a task forever if you forget to tell it to stop. Unfortunately, when you write WHILE or REPEAT loops it's rather easy to forget. Look at Example 8-5 and imagine what happens if you forget the statement $I := I + 1$. Don't forget: *Make sure that repeating the task specified in a WHILE loop will eventually cause the condition to become FALSE. Similarly, make sure that repeating the task specified in a REPEAT loop will eventually cause the condition to become TRUE.*

Lest our warnings of the previous paragraph put you off from using WHILE or REPEAT at all, let us offer you a word of reassurance. Your task will not really be repeated "forever," but only until some extraordinary action is taken. In the early days of computers, the computer operator would have to eventually push the STOP button. Modern computing systems protect themselves against infinite loops by stopping your job after a certain amount of time has elapsed. (You might say that the sorcerer eventually turns up and rescues the apprentice.) However, if there's a WRITELN in the loop, you may generate a huge (and expensive) stack of paper before your job is kicked out of the computer.

Here's a more subtle example of how you might accidentally write an infinite loop. Suppose you want to print a table of areas of circles where the radii are 0.5, 1.0, 1.5,...,5.0. You might write:

```
RADIUS := 0.5;  
REPEAT  
  AREA := PI * RADIUS * RADIUS;  
  WRITELN (RADIUS, AREA);  
  RADIUS := RADIUS + 0.5  
UNTIL RADIUS = 5.5
```

This looks superficially reasonable. However, remember that *REAL arithmetic in a computer is not exact, nor are most REAL numbers represented exactly*. The condition $\text{RADIUS} = 5.5$ may not ever be exactly satisfied. Hence, infinite loop! How might we fix this example? Our first thought might be to rewrite the condition as $\text{RADIUS} > 5.0$. This has gotten rid of the infinite loop, but we probably have fallen into another trap. Because of the inexact arithmetic, the condition may be satisfied *before* we do the computation for 5.0. That is, the value stored in RADIUS at this point may actually look like, say, 5.000001. Similarly, if we change the condition to $\text{RADIUS} < 5.5$, the task may be repeated for $\text{RADIUS} = 5.5$. What we need to do is to remember the inexactness of REAL computations and allow a *small tolerance* in writing comparisons. A condition that will have just the effect we want is, for example, $\text{RADIUS} > 5.1$.

To summarize: *In writing conditions involving REAL values, remember to allow tolerances for inexactness.*

8-10 A PROGRAMMING EXAMPLE

Suppose that the local county tax collector needs a program to calculate everyone's property taxes, given assessed valuations and tax rates. The tax collector certainly would not want to specify the total number of property owners, so this is clearly an example where we want to use a WHILE or REPEAT to test for a special value put at the end of the data.

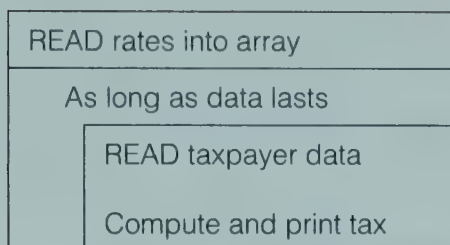
As a first step, we must firm up the *specifications* for the program. Exactly what data needs to be input? Is it available in any particular format? Exactly what needs to be computed? In what format is the output desired?

First, let us consider the input for the program. Perhaps it should consist of a list of values

TAXPAYER	VALUATION	RATE
----------	-----------	------

with the values for each taxpayer typed on a data card (or as a line of terminal input). Then we need only read a line and multiply VALUATION times RATE to get the tax owed by that taxpayer. However, we should look more closely at this approach. Does it really make sense to type in the rate for each taxpayer? (Think of the possibility for errors in typing!) So we should consult the tax collector: "Are all rates perhaps the same?" He tells us, "No, but the rates for everyone in the same township are the same." Here's a clear simplification. The taxpayer data need only include the township number. We can read the rates for the different townships into an array at the beginning of our program and then use the same trick as in Section 7-8 to multiply the VALUATION by the correct RATE, pulled out of the array. Our overall program structure is shown in Figure 8-5.

Next let us develop a more detailed specification of the data. Although the number of townships in the county is unlikely to change, we might as well keep things general by reading in this number as a variable. (Besides, the



LOOP CONTROL
WITH WHILE
OR REPEAT
STATEMENTS

Figure 8.5

facility to set a small number of townships will make testing the program easier.) We'll have to know a maximum, though, to set the array size in the declaration. The final input data specification might then look like:

Specification of Input

Number of townships (INTEGER variable named N)

List of rates (array of REALs named RATE)

Unknown number of tax-record cards in format:

TAXPAYER TOWNSHIP VALUATION

where TAXPAYER is an eight-digit INTEGER

TOWNSHIP is an INTEGER less than 100

VALUATION is a REAL

Last card in data deck contains 0 as the taxpayer number, to signal the end of the data

Notice that we have included variable names here as an aid to keeping track of the variables in later discussions.

Specification of Output

An appropriately headed table should be printed with columns:

TAXPAYER VALUATION TOWNSHIP TAX OWED

Notice that we have printed out all the pertinent information here so that we, the tax collector, and the taxpayer can check for peculiarities. (In general, we would probably want to print the rate also, but it is not done here because extra-wide lines do not fit well into book format.)

Now let us expand our flowchart into one with more detail. First the rectangle for reading the rates is shown in Figure 8-6.

READ Rates

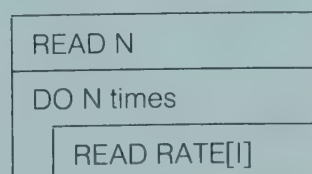


Figure 8.6

Now we need to print table headings (Figure 8-7), a step which we neglected in our first flowchart.

Print headings

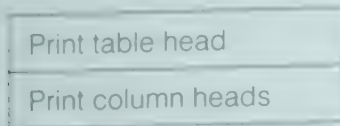


Figure 8.7

Now for the compute-and-print loop (Figure 8-8).

Compute-and-Print Loop

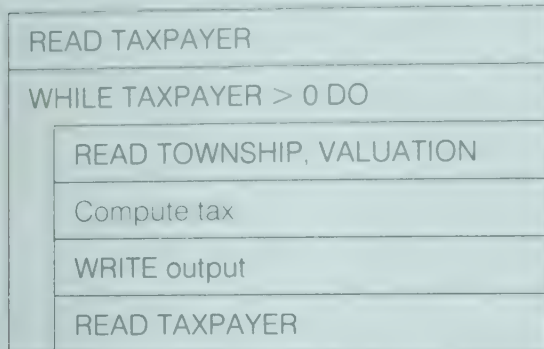


Figure 8.8

Notice that we now have three rectangles to describe the program. They fit together as shown in Figure 8-9.

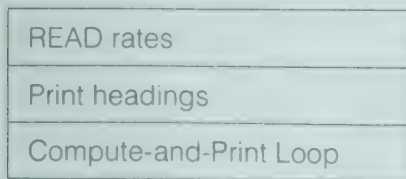


Figure 8.9

By *labeling* the detail boxes as we have and showing how they fit together by means of the higher level chart, we avoid generating gigantic, crowded detail charts of a whole program in one piece. This makes the structure of the program much easier to grasp.

At this point we can write the following program. In fact, after so carefully detailing what we want to do, we can jot the program down in about five minutes, plus a bit of extra time to figure out the table spacing.

```

PROGRAM TAXCOMP ( INPUT, OUTPUT );
VAR I, N, TAXPAYER, TOWNSHIP: INTEGER;
    VALUATION, TAX: REAL;
    RATE: ARRAY [1..99] OF REAL;
BEGIN
    READ (N);
    FOR I := 1 TO N DO
        READ (RATE[I]);
    Writeln ( '                PROPERTY TAXES' );

```

```

WRITELN ( '          *****' );
WRITELN;
WRITELN ( ' TAXPAYER   VALUATION   TOWNSHIP' );
WRITELN ( '      TAX OWED' );
READ (TAXPAYER);
WHILE TAXPAYER > 0 DO
    BEGIN
        READ (TOWNSHIP, VALUATION);
        TAX := RATE[TOWNSHIP] * VALUATION;
        WRITELN (TAXPAYER:8, VALUATION:15:2,
            TOWNSHIP:10, TAX:16:2);
        READ (TAXPAYER)
    END
END.

```

We then test the program with the following set of data:

```

2
0.04    0.05
12345678    1  9743.50
87654321    2  20550.00
98765432    1  30100.10
0

```

And obtain the following output:

```

          PROPERTY TAXES
*****
TAXPAYER   VALUATION   TOWNSHIP   TAX OWED
12345678    9743.50        1      389.74
87654321   20550.00        2     1027.50
98765432   30100.10        1     1204.00

```

8-11 EXERCISES

- 8-1.** Using an INTEGER variable to hold an eight-digit taxpayer identification number (as in Section 8-10) won't be feasible if your computer can't handle integers that large. The largest integer that one can use in a Pascal program is an implementation-dependent constant named MAXINT. You can use MAXINT in a program like any other constant, only, of course, you cannot declare it. Find the value of MAXINT for your computer.
- 8-2.** Rewrite Example 7-5 using WHILE loops instead of FOR loops.
- 8-3.** Rewrite the program of Section 7-8 to use a WHILE loop instead of reading and processing exactly $3 \cdot N$ grade cards. Also change the specification of the input data accordingly.
- 8-4.** The following program segment is supposed to find the largest value in $A[1..50]$ and store the value in MAX. The index of the largest value is to be stored in IMAX. Does the program actually do this?

```

MAX := A[1];
IMAX := 1;
I := 2;
REPEAT
    WHILE A[I] < MAX DO
        I := I + 1;
    WHILE A[I] > MAX DO
        BEGIN
            MAX := A[I];
            IMAX := I;
            I := I + 1;
        END
UNTIL I = 50

```

8-5. The array PROD[1..50] has been initialized so that PROD[I] contains the value I. At what value of I does the loop terminate in each of the following examples?

(a) I := 2;
 WHILE PROD[I] < 30 DO
 BEGIN
 PROD[I] := PROD[I-1] * 3;
 I := I + 1;
 END

(b) I := 2;
 REPEAT
 PROD[I] := PROD[I-1] * 3;
 I := I + 1;
UNTIL PROD[I] > 30

(c) I := 1;
 REPEAT
 I := I + 1;
 PROD[I] := PROD[I-1] * 3
UNTIL PROD[I] > 30

8-6. In the segments below write the condition necessary to sum the integers from 1 to 10.

```

I := 0;
SUM := 0;
WHILE _____ DO
    BEGIN
        SUM := SUM + I;
        I := I + 1;
    END

```



```

I := 0;
SUM := 0;
WHILE _____ DO
    BEGIN
        I := I + 1;
        SUM := SUM + I
    END

```

```

I := 0;
SUM := 0;
REPEAT
    SUM := SUM + I;
    I := I + 1
UNTIL _____

```

```

I := 0;
SUM := 0;
REPEAT
    I := I + 1;
    SUM := SUM + I
UNTIL _____

```

- 8-7.** Write a program that computes squares and cubes of the positive integers 1, 2, 3,.... The output should be an appropriately headed table with columns I, I², I³. [Column headings might, for example, be I, SQUARE(I), CUBE(I).] Use a WHILE or REPEAT to stop the computation as soon as I³ becomes greater than 10,000.
- 8-8.** A meteorologist keeps records of the weather for each month as a deck of punched cards. The first card of the deck gives the number of days of the month.
The following cards give the rainfall (in inches), low and high temperatures (both in degrees Fahrenheit), and pollution count for each day of the month. Example:

31			
3	50	67	9
1	53	72	11
4	52	74	7
...			

Write a Pascal program that finds the first rainy day of the month (if any). Print the date and the amount of rainfall if one is found. Otherwise print an appropriate message.

- 8-9.** Twenty-five integers are stored in a one-dimensional array A. Exactly one of these numbers is zero. Write a program segment that finds the zero and replaces it with the number 1.

- 8-10.** The Fibonacci sequence is a sequence of integers that begins 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... Each number beyond the second is the sum of the previous two numbers. More precisely, the numbers of the sequence, denoted by F_k , are defined by $F_1 = 1$, $F_2 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for $k > 2$. Write a program that has the following input and output.

Input

The input will be pairs of integers M and N ($M \leq N$), both less than $F_{25} = 75,025$. The value of M will precede the value N on each input line.

Output

- (a) Print the values M and N.
 - (b) Print all Fibonacci numbers between M and N. That is, print all F_k such that $N < F_k < M$.
 - (c) Print the number of Fibonacci numbers between M and N. Thus if $M = 8$ and $N = 3$, this number would be 1, since the only Fibonacci number between 3 and 8 is 5.
 - (d) Print the number of *even* Fibonacci numbers between M and N.
 - (e) Print the number of *odd* Fibonacci numbers between M and N.
- 8-11.** Write an interactive program to carry out the task of Exercise 6-7. Prompt for employee number, wage, and hours worked. Instead of reading in the number of employees, ask the user if he or she is done by a message such as:

FINISHED? IF YES, TYPE "2" ELSE TYPE "1"

SUMMARY OF CHAPTER 8

The WHILE statement repeatedly executes a task *as long as* a given condition is satisfied. The REPEAT statement repeatedly executes a task *until* a given condition is satisfied.

Conditions are specified by boolean expressions, which evaluate to either TRUE or FALSE. Boolean expressions can simply be BOOLEAN variables or can be constructed as comparisons between numerical quantities. The comparison operators are $=$, $>$, $<$, $<=$, $>=$. Furthermore, more complicated expressions can be built up from simpler ones by using the boolean operators AND, OR, NOT.



THE IF AND CASE STATEMENTS

9-1 INTRODUCTION

So far we have seen several ways in which the steps of a program are controlled. At the beginning of this book we saw the simplest case, in which the statements in a program are executed one by one with no repetition. We then looked at the case in which the FOR statement is used to repeat a statement (or compound statement) a given number of times. We also saw that the WHILE statement or the REPEAT statement can be used to repeat a group of statements according to the truth or falsity of a given condition. In all these cases, the same set of statements is executed every time the program is run, although a loop may be executed different numbers of times. There are many situations, however, in which a program should perform *different* computations under different circumstances. We saw an example of this in the last chapter, where we wanted to print out different messages depending upon whether the data included at least one entry for a person with AGE ≥ 65 . As another example, a program to compute income taxes must proceed differently, depending on whether the taxpayer is married or single and what tax bracket he or she is in. There are many other examples in which a computer needs to take a different path of action depending on unforeseeable circumstances. A spacecraft landing on Mars will take different actions depending on temperature and pressure measurements. Insurance premiums will be computed differently for people in different age groups. In this chapter we shall study the Pascal language statements that can be used to direct a computation through different paths, depending on specified conditions.

9-2 THE IF STATEMENT

The IF statement in Pascal is very much like English sentences such as "If there are fresh strawberries in the supermarket, then buy two quarts; otherwise, buy a can of peaches." In a Pascal-like statement, we write:

```
IF there are fresh strawberries
  THEN buy two quarts
  ELSE buy one can of peaches
```

Notice that "there are fresh strawberries" is a *condition*, just like those that appear in WHILE or REPEAT statements. If the condition is true, we take one action; if it is false, we take a different action. If you compare the Pascal-like statement with the English sentence, you see that the difference is only that Pascal uses ELSE instead of "otherwise," probably because "otherwise" is too tedious to write.

To be more precise, an IF statement in Pascal is of the form

```
IF boolean expression
  THEN statement
  ELSE statement
```

When an IF statement is executed, first the boolean expression is evaluated. If its value is true, the statement following THEN will be executed. If the value of the boolean expression is false, the statement following ELSE will be executed. In either case, the next statement executed will be the one following the IF statement.

EXAMPLE 9-1

```
IF INCOME > 10000.00
  THEN TAX := 0.35 * (INCOME - 10000.00) + 2000.0
  ELSE TAX := 0.2 * INCOME;
Writeln(TAX)
```

In this example, if INCOME exceeds 10000.00, TAX will be computed according to one formula; and if INCOME is less than or equal to 10000.00, TAX will be computed according to a different formula. In either case, the next statement executed will be the WRITELN.

Notice the punctuation in Example 9-1. There is only one semicolon, the one at the end of the third line. This is because the IF statement is just one long statement. We just need the single semicolon to separate the IF statement from the WRITELN statement.

Sometimes we need to use compound statements to describe the actions to be taken, as the following example illustrates.

```
IF INCOME > 10000.00
THEN
  BEGIN
    TAX := 0.35 * (INCOME - 10000.00) +
      2000.00;
    WRITELN('TAX BRACKET IS 35%; TAX IS',
      TAX:15:2)
  END
ELSE
  BEGIN
    TAX := 0.2 * INCOME;
    WRITELN('TAX BRACKET IS 20%; TAX IS',
      TAX:15:2)
  END
END
```

Notice that, as usual, there are semicolons separating the statements within each compound statement.

All of the examples we have shown you so far contain IF statements of the form

```
IF condition
  THEN statement
  ELSE statement
```

Actually, in Pascal it is legal not to include the ELSE part of the IF statement. An IF statement of the form

```
IF condition
  THEN statement
```

is equally valid. In this case, if the condition is true then the specified action will be taken. Otherwise, no action at all will be taken. Example 9-2, which we shall study next, shows how IF statements without an ELSE clause can arise very naturally.

Example 9-2 will furthermore provide you with an illustration of the use of IF statements within a complete program. This program converts an integer into its representation in Roman numerals. We remind the reader that in the Roman representation of integers, M stands for 1000, D stands for 500, C stands for 100, L stands for 50, X stands for 10, V stands for 5, and I stands for 1. Notice that in the representation of any integer there is at most one D, one L, and one V, while there can be many M's, C's, X's, and I's. In the program, a character (Roman numeral) is printed, following which its decimal equivalent is subtracted from the number which is being printed. Note that an IF statement is used when a character (e.g., D) can occur at most once, and a WHILE statement is used when a character (e.g., M) can occur more than once.

Notice another feature in this program. We need to print the characters one at a time as we work them out. Using `WRITELN` for this is not satisfactory, since it would print each character on a new line. The solution is to use `WRITE`, which prints successive items all on the same line.

EXAMPLE 9-2

```

PROGRAM ROMAN( INPUT, OUTPUT );
(* CONVERTS AN INTEGER N TO ROMAN NUMERAL FORMAT *)
VAR N: INTEGER;
BEGIN
  READ(N);
  WRITE(N, ' IN ROMAN NUMERALS IS ');
  WHILE N >= 1000 DO
    BEGIN (* PRINT 'M' FOR EACH 1000 *)
      WRITE('M'); N := N - 1000 (*REDUCE N *)
    END;
  IF N >= 500 THEN
    BEGIN (* PRINT 'D' FOR THE 500 *)
      WRITE('D'); N := N - 500 (* REDUCE N *)
    END;
  WHILE N >= 100 DO
    BEGIN (* PRINT 'C' FOR EACH 100 *)
      WRITE('C'); N := N - 100 (* REDUCE N *)
    END;
  IF N >= 50 THEN
    BEGIN (* PRINT 'L' FOR THE 50 *)
      WRITE('L'); N := N - 50 (* REDUCE N *)
    END;
  WHILE N >= 10 DO
    BEGIN (* PRINT 'X' FOR EACH 10 *)
      WRITE('X'); N := N - 10 (* REDUCE N *)
    END;
  IF N >= 5 THEN
    BEGIN (* PRINT 'V' FOR THE 5 *)
      WRITE('V'); N := N - 5 (* REDUCE N *)
    END;
  WHILE N > 0 DO
    BEGIN (* PRINT 'I' FOR EACH 1 *)
      WRITE('I'); N := N - 1
    END
  END.

```

In Example 9-2 we probably would want to put a limit—say, 5000—on the integers that the program should handle. We also, of course, can't handle negative integers or zero. We mentioned in an earlier chapter that checks for

valid data should be built into every program. The way to do this is with IF statements. We might, therefore, change the program ROMAN as follows:

```
PROGRAM ROMAN2( INPUT , OUTPUT ) ;
VAR N: INTEGER ;
BEGIN
  READ(N) ;
  IF (N > 5000) OR (N < 1)
    THEN (* DATA FAILS VALIDITY TEST *)
      Writeln(N, ' IS NOT VALID DATA')
    ELSE (* CONVERT TO ROMAN NUMERAL *)
      BEGIN (* COPY STATEMENTS FROM ROMAN ,
              EXCEPT FOR READ *)
      END
  END .
```

In essence, the structure of this program is

```
  READ data
  IF data is bad
    THEN print a message
  ELSE do the computation
```

Clearly a computation of arbitrary complexity can follow the ELSE in such a program. In our example (program ROMAN2), several WHILEs and IFs follow the ELSE.

On the other side of the coin, we may wish the action specified in either the ELSE clause or the THEN clause to be "Do nothing." As we have noted, this is easy to take care of in the ELSE clause, since that clause can simply be omitted. In Pascal there is a way to specify "do nothing" after a THEN or whenever else we might find it useful. All we have to do is not include any command. To make the absence of a command legal, the so-called *null statement* is formally included in the Pascal language. For example:

```
IF GRADE > 50
  THEN
  ELSE Writeln ('YOU FLUNK.')
```

Here we specify that nothing is to be done if the grade is acceptable (>50). Of course, in a case like this it is probably easier to turn the condition around and write

```
IF GRADE <= 50
  THEN Writeln ('YOU FLUNK.')
```

```
or
IF NOT (GRADE > 50)
  THEN Writeln ('YOU FLUNK.')
```

But remember that in any situation where it is more natural to write a condition in such a way that it leads to a null statement after the THEN, it is perfectly legal to do so.

Being aware of the existence of the null statement will help you better understand what happens when you include a semicolon where it doesn't belong. Suppose that in the preceding example you put a semicolon after the THEN:

```
IF NOT (GRADE > 50) THEN;  
    WRITELN ('YOU FLUNK.')
```

The IF statement (which is terminated by the semicolon) now has no effect whatsoever, and YOU FLUNK. is *always* printed.

But now consider the following compound statement.

```
BEGIN  
    WRITE ('THIS IS ');  
    WRITELN ('A TEST');  
END
```

Oops! We put a semicolon after the second WRITELN. We don't have to panic, though, since a null statement will be assumed before the END, and no error will actually occur.

9-3 FLOWCHARTS WITH BRANCHING

A new flowchart picture is needed to indicate that one of two alternative actions will be taken, depending upon whether a condition is true or false. The picture is shown in Figure 9-1.

Notice that, as for other types of statements, the picture representing an IF statement is a rectangle. Furthermore, the action taken in each case is also represented by a rectangle, since the action will be specified by a statement. For program ROMAN2, the highest level flowchart would be as shown in Figure 9-2.

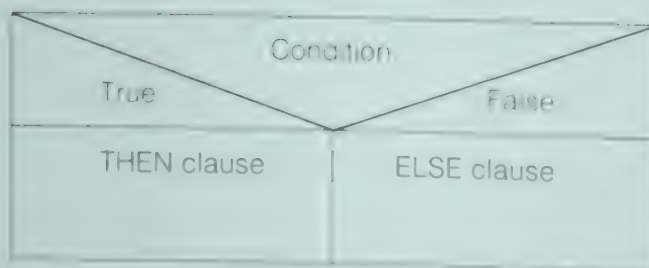


Figure 9-1

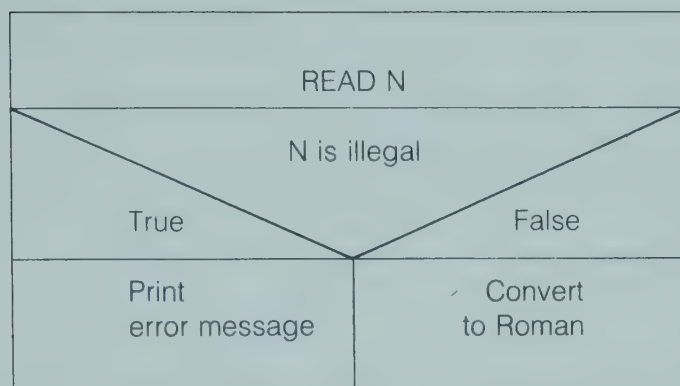


Figure 9-2

The rectangle labeled "Convert to Roman" can be further subdivided. See Figure 9-3.

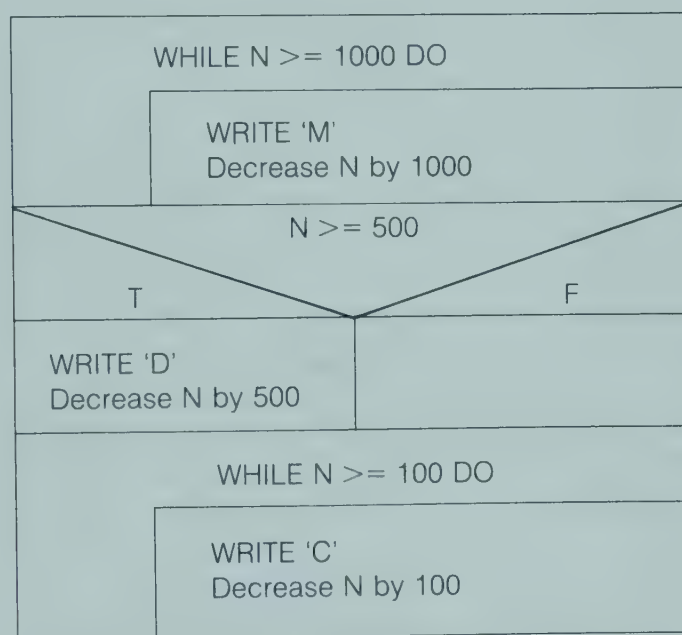


Figure 9-3

(We will leave it as an exercise for the reader to complete the flowchart for ROMAN2.) Notice that we cannot omit the ELSE path in the flowchart as we can in the program. So we simply leave it empty to signify that no action is taken in this case. Similarly, if we want to take no action when the condition is true, we can leave the THEN clause box empty. However, we *cannot* omit the THEN in our program, but must follow it by the null statement, as we have discussed.

9-4 NESTING IF STATEMENTS

You have already seen an example of one IF statement included within another IF statement. (See program ROMAN2.) Just as the statements within a FOR statement may include other FORs, so the statements within

an IF statement may include other IFs—and these may contain still further IFs, and so forth. This facility allows us to build up quite complicated structures of actions based on conditions and to program them clearly and succinctly. Thus, suppose X, Y, and Z are conditions, and there are four different computations that might be carried out, depending upon the truth or falsity of X, Y, and Z. Rules for making the decision are as follows:

- If X and Y are true, do computation 1.
- If X is true and Y is false, do computation 2.
- If X is false and Z is true, do computation 3.
- If X is false and Z is false, do computation 4.

One way to program this would be to use a sequence of IF...THEN statements with compound conditions—almost literally transcribing the English-language description of what's to be done. The resulting program would read:

```
IF X AND Y THEN ...;      (* COMPUTATION 1 *)
IF X AND NOT Y THEN ...;  (* COMPUTATION 2 *)
IF NOT X AND Z THEN ...;  (* COMPUTATION 3 *)
IF NOT X AND NOT Z THEN ... (* COMPUTATION 4 *)
```

But notice the inefficiency of this version. The conditions for each IF must be evaluated. This means, for example, evaluating X *four times*. If X involves a complicated arithmetic expression, this can be very time-consuming.

To construct a more efficient program we nest the IF statements so as to reduce the total number of condition evaluations. Let us see what happens for this particular example. First, we see that all the conditions involve X. Furthermore, when X is true, a test of Y determines whether we do computation 1 or computation 2. Similarly, when X is false a test of Z determines whether we do computation 3 or computation 4. From these observations, we can construct a flowchart (Figure 9-4) describing the rules. In this chart we have used encircled numbers to indicate the four different computations.

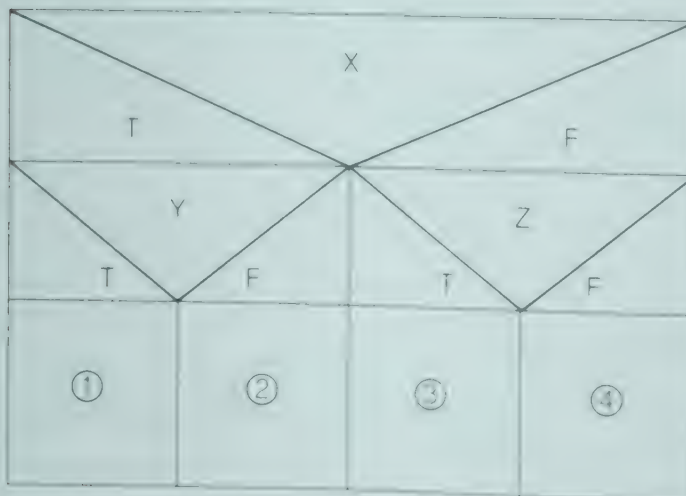


Figure 9-4

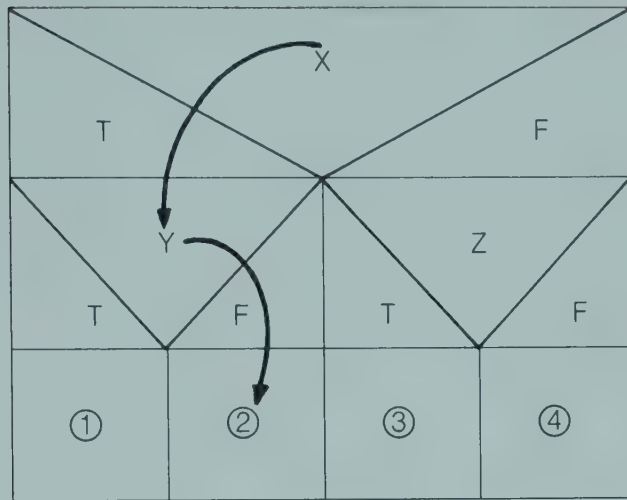


Figure 9-5

By examining the different paths through this flowchart, you can easily check that the original specification is satisfied. For example, follow the arrows as shown in Figure 9-5.

You see that when X is true and Y is false (i.e., IF X AND NOT Y) then computation 2 is carried out. You can check the other paths yourself.

Once a flowchart with nesting has been constructed, it is a trivial matter to convert it into a nested IF statement. Here is the result.

```

IF X
  THEN
    IF Y
      THEN ①
      ELSE ②
    ELSE
      IF Z
        THEN ③
        ELSE ④

```

Notice that in this program, each condition is evaluated only once. In general, you will find that nesting IF statements has the effect of eliminating—or at least decreasing—the repetitive evaluation of conditions.

9-5 THE CASE STATEMENT

The IF statement allows a program to *branch*; i.e., to follow one of two paths depending upon whether a condition is true or false. In many situations, however, a two-way branch is not a natural one. In computing our income tax, the computation is not a simple two-way choice depending upon whether income is more or less than 10000, as in Example 9-1. Instead, there are a dozen or so ranges of income for which the computation is different.

We can, of course, handle a multiple branch by using nested IF statements. As an example, here's a program that uses a nested IF statement to read an integer from 1 to 7 and print out the name of the corresponding day of the week.

EXAMPLE 9-3

```

PROGRAM NAMEDAY ( INPUT, OUTPUT );
VAR N: INTEGER;
BEGIN
  READ (N);
  WRITE ( 'DAY', N:2, ' IS ');
  IF N = 1 THEN WRITELN ( 'SUNDAY.' )
  ELSE IF N = 2 THEN WRITELN ( 'MONDAY.' )
  ELSE IF N = 3 THEN WRITELN ( 'TUESDAY.' )
  ELSE IF N = 4 THEN WRITELN ( 'WEDNESDAY.' )
  ELSE IF N = 5 THEN WRITELN
    ( 'THURSDAY.' )
  ELSE IF N = 6 THEN WRITELN
    ( 'FRIDAY.' )
  ELSE IF N = 7 THEN WRITELN
    ( 'SATURDAY.' )
END .

```

Notice how complicated this looks. And how easy it would be to get confused and forget an ELSE or make some other careless error. What we would like to do is to structure programs like this *not* as a sequence of two-way branches, but as a single seven-way branch. It is much more natural to analyze this example as follows: There are *seven cases*—depending upon the value read. According to which of the seven cases holds true, an appropriate message should be printed. The Pascal facility to let us do this is called a CASE statement. For the example we are working on, we declare an INTEGER variable N which is to take on a value from 1 to 7. Then we write a CASE statement:

```

CASE N OF
  1: WRITELN ( ' SUNDAY.' );
  2: WRITELN ( ' MONDAY.' );
  3: WRITELN ( ' TUESDAY.' );
  4: WRITELN ( ' WEDNESDAY.' );
  5: WRITELN ( ' THURSDAY.' );
  6: WRITELN ( ' FRIDAY.' );
  7: WRITELN ( ' SATURDAY.' )
END

```

When the computer executes this statement, it first looks at the value of N. Then it carries out the action specified following that value in the list. Thus, if N = 2, the action taken is to execute WRITELN (' MONDAY.').

The *N* is called the *case index*. In general, the case index may be an expression, which should evaluate to one of the specified values, or *case constants*. What happens when the case index does not evaluate to one of the listed case constants is implementation-dependent. Notice that the CASE ends with an END so that the compiler can tell when the list ends.

We have rewritten program NAMEDAY using a CASE statement, so that you can see how much more intelligible it looks. Here it is:

EXAMPLE 9-4

```

PROGRAM DAY ( INPUT, OUTPUT );
VAR N: INTEGER;
BEGIN
  READ (N);
  WRITE ( 'DAY', N:2, ' IS' );
  CASE N OF
    1: WRITELN ( ' SUNDAY.' );
    2: WRITELN ( ' MONDAY.' );
    3: WRITELN ( ' TUESDAY.' );
    4: WRITELN ( ' WEDNESDAY.' );
    5: WRITELN ( ' THURSDAY.' );
    6: WRITELN ( ' FRIDAY.' );
    7: WRITELN ( ' SATURDAY.' )
  END
END .

```

As you would expect of Pascal, the statements specifying the actions in each case can be compound to specify complicated actions or null, if we wish to do nothing. As a convenience, Pascal also allows several cases to be grouped together if the same action is to be taken. Thus if *N* again represents a day of the week, you might program yourself:

```

CASE N OF
  1: BEGIN
        Go to church; do homework
      END
  2, 3, 4, 5, 6: Go to school;
  7:
END

```

The flowchart picture for a CASE statement is just a small variation on that for an IF. For example, a four-way branch would be pictured as shown in Figure 9-6.

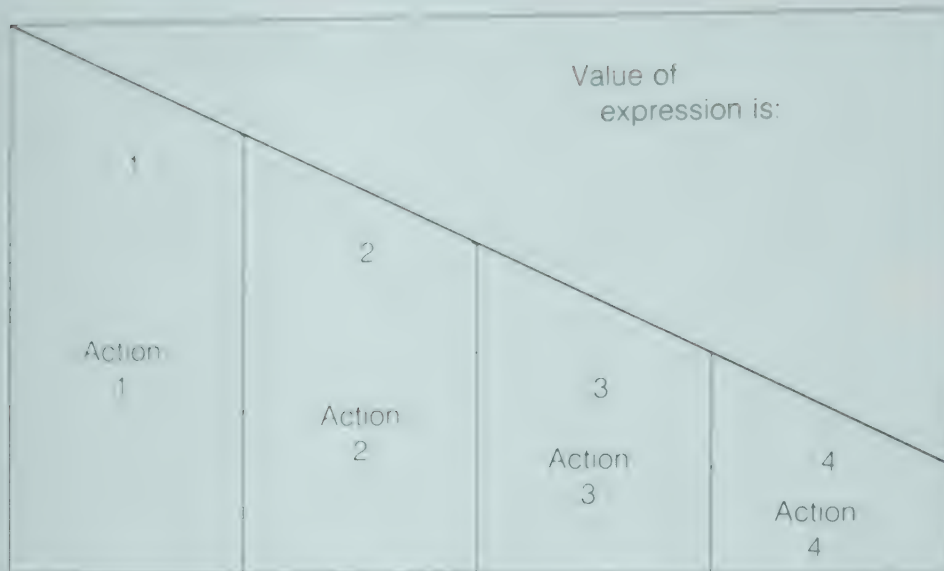
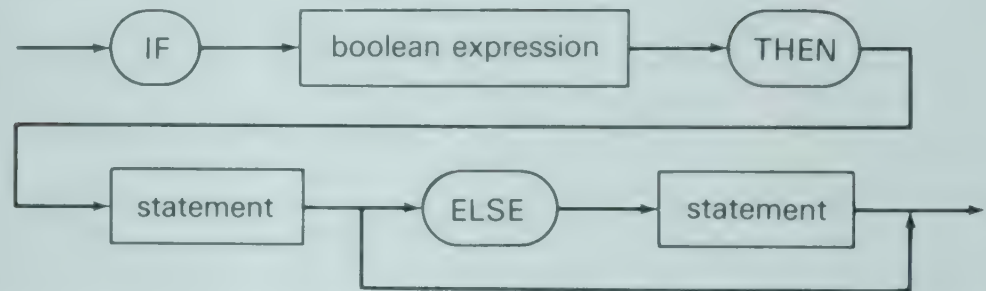


Figure 9-6

9-6 SYNTAX

The syntax for the *IF* statement is



This shows clearly that the ELSE clause may be omitted, but the THEN clause must always be included (although the statement following the THEN may be null).

There is one situation in which the syntax of the IF statement appears to be ambiguous. This is in the construct

```

IF EXPRESSION
  THEN IF ... THEN ... ELSE ...

```

The problem is, does the ELSE go with the first IF or the second IF? That is, to picture the choice via indenting, do we mean

```

IF ...
  THEN
    IF ... THEN ...
  ELSE ...

```

or do we mean

```

IF ...
  THEN
    IF ... THEN ...
    ELSE ...

```

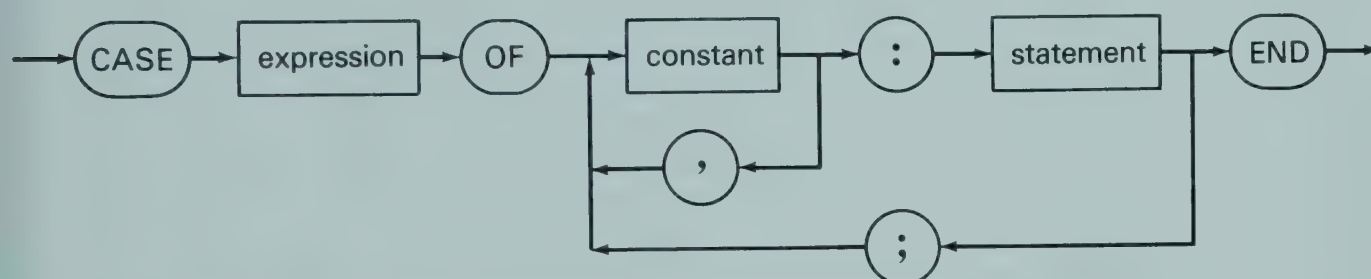
Notice that both interpretations are consistent with the syntax diagram. The ambiguity is resolved by assuming that the ELSE goes with the *most recent* (thus the second) IF. Formally, the construct is defined to be equivalent to:

```

IF EXPRESSION
  THEN
    BEGIN
      IF ... THEN ... ELSE ...
    END

```

The syntax of the *CASE statement* is, as you have seen, rather more complicated:



The “constants” here are, of course, the various allowed values of the “expression,” the values that label the statements to be executed under various conditions. Notice the possibility of having several constants in succession (separated by commas) labeling a statement. This corresponds to the situation, which we discussed earlier, where the same action is to be taken for more than one value of the expression.

9-7 PROGRAMMING

HINTS: EFFICIENCY AND TESTING

9-7-1 Efficient Nesting

In Section 9-4 we pointed out that it is easy to unthinkingly write IF statements that require far more evaluations of boolean expressions than are necessary. The point we made there was that by *nesting* IF statements we can generally obtain a more efficient program than we would obtain with a sequence of separate IF statements.

A further point that we wish to make here is that the *order* in which conditions are tested can sometimes make a large difference in efficiency. To see why this is so, consider the following example.

EXAMPLE 9-5

```

IF GRADE >= 95
  THEN WRITELN ('STUDENT GETS A+')
ELSE IF GRADE < 50
  THEN WRITELN ('STUDENT FLUNKS')
ELSE IF GRADE < 65

```



```

THEN WRITELN ( 'STUDENT BORDERLINE' )
ELSE WRITELN ( 'STUDENT OK' )

```

Suppose that nearly all students are OK, with grades between 65 and 95. Then, when this statement is executed, three IF statements, involving evaluation of three conditions, must be processed nearly every time. It is much more efficient to first evaluate the condition that is most likely to be true, so that the rest of the tests may be skipped. Furthermore, we would probably want to put the next most likely case second, and so forth. Following this advice, we might rewrite Example 9-5 as follows:

```

IF ( GRADE >= 65 ) AND ( GRADE < 95 )
  THEN WRITELN ( 'STUDENT OK' )
ELSE IF GRADE >= 95
  THEN WRITELN ( 'STUDENT GETS A+' )
ELSE IF GRADE >= 50
  THEN WRITELN ( 'STUDENT BORDERLINE' )
  ELSE WRITELN ( 'STUDENT FLUNKS' )

```

Notice that by rearranging the order of the tests we have been obliged to evaluate two inequalities in the first condition instead of one. If the "OK" case were just a little bit more probable than the others, the rearrangement would probably be counterproductive. Thus, although the "BORDERLINE" case is more likely than the "A+" case, we opted not to arrange the tests in that order. (The reader should rewrite the IF statement for that order of tests to see why.)

9-7-2 Moving Tests Outside of Loops

You should recall that earlier (Section 7-7) we emphasized the importance of moving computations outside loops, whenever possible, to avoid repeating identical computations. In this regard, you should keep in mind that the evaluation of a boolean expression is a computation that can be quite expensive. Here's an example of a computation that looks very natural but is, in fact, inefficient.

```

FOR I := 1 TO 1000 DO
  IF TEST = 1
    THEN X(I) := A(I) + B(I)
    ELSE X(I) := A(I) - B(I)

```

The expression `TEST = 1` is evaluated 1000 times—exactly 999 times more than is necessary! To fix this we write:

```

IF TEST = 1
  THEN
    FOR I := 1 TO 1000 DO
      X(I) := A(I) + B(I)

```

ELSE

FOR I := 1 TO 1000 DO

X(I) := A(I) - B(I)

This saves a lot of computation at the tiny expense of putting an extra FOR statement in our program. (Of course, we can't move the IF outside the loop in this way if the condition depends on I.)

9-7-3 Testing Programs with Branches

Finally let us look at the complexities that branching can add to the problems of testing and debugging a program. The most obvious point is that *the code contained in every THEN clause and every ELSE clause should be tested*. Suppose that we have IF statements as shown in Figure 9-7. We must make sure that we test the program with enough sets of data so that all four program segments (1, 2, 3, and 4) have at some time been executed. If we want to make sure which branches have been executed on any particular run—which is important information for debugging a program—we should put a WRITELN with an appropriate message in each code segment. (Students sometimes waste enormous amounts of time in looking for program bugs by *assuming* which branches of the program were executed.)

If you think about it, you will realize that it is not likely that a program with an IF statement in it can be effectively tested by one set of data. Even when an IF statement exists only to provide a check for bad data, we would want to input some bad data to make sure that the validity check works.

A rather subtle difficulty with program testing is that executing every individual segment of the program may not be enough. Consider the example above more closely. We might run the program with one set of data for which X and Y are both true. This executes 1 and 3. Then we might run the program again with data for which X and Y are both false. This executes 2 and 4. However, we *cannot* conclude at this point that the program is well tested, since there may be some sort of interaction between these program segments that causes a failure when X is true and Y is false. (Segment 1 may change a variable that segment 4 uses, for example.) Another possible goal is to *test all possible paths through the program*, that is, to test all possible sequences of statements that might be executed under any possible set of circumstances. In the case of our current example, we would want to design tests of the *four*

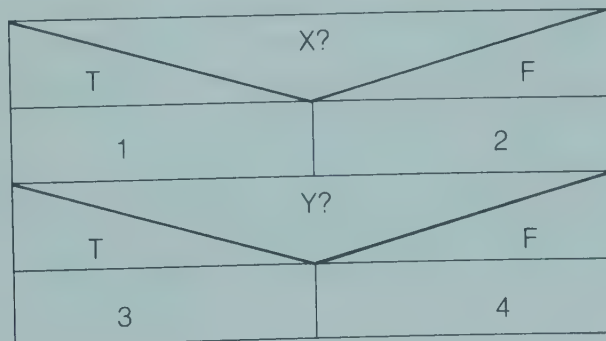


Figure 9-7

paths 1, 3; 1, 4; 2, 3; and 2, 4. In general, there may be an infinite number of possible sequential paths through a program. Consider a loop that may be executed any number of times, depending on a number read. One path has one loop execution, a second has two loop executions, and so forth. One feels—and rightly—that one will gain very little extra knowledge about a program by executing a simple loop many different numbers of times. But checking the various paths through a sequence of IF statements can be important.

One has to use some judgment as to what paths need testing; i.e., when the behavior might be really different along different paths. More importantly, the goal of executing all possible paths is still inadequate. A path may appear correct for one set of data but fail on some other set. The statements along a path may execute correctly, but a wrong branch may be taken for some set of data. (Having nice smooth roads doesn't help if you take the wrong turn!) Furthermore, some paths that should be in the program (e.g., statements to handle special cases or validity checks) may be *missing*; in this case a program can be fatally flawed and path testing will never discover the fact. Generally, one attempts to combine the idea of path testing with the idea that the program should be tried out for a broad sample of possible input data, including special cases, bad data, etc.

9-8 A PROGRAMMING EXAMPLE: SORTING

Most computerized data processing would be hopelessly inefficient if the data were not sorted first. *Sorting* means to rearrange a set of data into some prescribed order, such as alphabetic or increasing numeric order. Your local bank, for example, almost surely sorts the list of daily transactions (deposits and withdrawals) before updating the accounts. This has the obvious effect of grouping together all the transactions for one account, so that accounts are processed one after the other in the order in which the computer has them stored. The gain in efficiency of this approach can be understood by comparing two shoppers. One proceeds up and down the supermarket aisles in order, picking up items as he or she comes to them. The second runs madly back and forth across the length of the store, visiting each aisle many times. This shopper is slavishly following a randomly ordered shopping list. The first shopper, on the other hand, has sorted (at least mentally) his or her shopping list to match the order in which items are placed in the store.

In summary, sorting is a critically important operation. At this minute, a significant fraction of the world's computing power is engaged in the **mundane work of sorting data**.

Suppose we are given a set of positive integers to sort into ascending order. For example, given

5, 10, 4, 7, 3, 1

we would want to put them into the order:

1, 3, 4, 5, 7, 10

We need to make this a bit more precise. Suppose the original, unsorted set of integers is read into an array named *A*. Thus, in our example, we would have 5 in *A*[1], 10 in *A*[2], etc. What we want to do is to rearrange these numbers, so that after the sort has been carried out we have 1 in *A*[1], 3 in *A*[2], etc.

Let us start with an elementary case. Suppose *A* has only two elements—how should we proceed? Clearly there are only two possibilities. If $A[1] \leq A[2]$, the numbers are already sorted and we need not do anything. Otherwise we simply *exchange* the two. A program statement to sort two elements is then:

```
IF A[1] > A[2]
  THEN
    BEGIN
      TEMP := A[1];
      A[1] := A[2];
      A[2] := TEMP
    END
```

This simple two-element comparison and exchange forms the basis of all computational methods (or *algorithms*) for sorting. The differences in the algorithms lie in the various ways in which the elements for comparison are chosen.

In this chapter we will look at what is probably the simplest of the sorting algorithms. This is *bubblesort*, a scheme in which adjacent elements in the array are successively compared and exchanged. Before writing a program, let us look at how the algorithm works on our example.

We start with the numbers:

5	10	4	7	3	1	compare and →	5	10	4	7	3	1	compare and →	4	10	7	3	1	exchange	4	10	7	3	1	→	5	4	7	3	1	→	5	4	7	3	1	→	5	4	7	3	1	→	5	4	7	3	1

After this *first pass* through the array, the number 10 is last in the list, as we want it to be. (It has, so to speak, “bubbled down” to the bottom. At the same time, the smaller numbers are “bubbling up” towards the top.) Since the last element is now in its proper place, the *second pass* requires comparison of one less pair of numbers. In detail

5}	→	4	4	4
4}		5}	5	5
7		7}	3	3
3		3}	→	7}
1		1	1}	→
				7
10		10	10	10

Now both 7 and 10 are correctly located, so on the *third pass*:

4}		4	4
5}		3	3
3}	→	5}	1
1		1}	→
			5
7		7	7
10		10	10

And on the *fourth pass*:

4}	→	3	3
3}		4}	1
1		1}	→
			4
5		5	5
7		7	7
10		10	10

And finally (*fifth pass*):

3}	→	1
1}		3
4		4
5		5
7		7
10		10

You should have enough experience by now to recognize the loop structure in this. If there are N elements in the array, there are $N - 1$ passes required (outer loop). And for the l th pass, $N - l$ comparison-and-exchange steps are required (inner loop). A flowchart for a program which reads and sorts an array of integers is given in Figure 9-8.

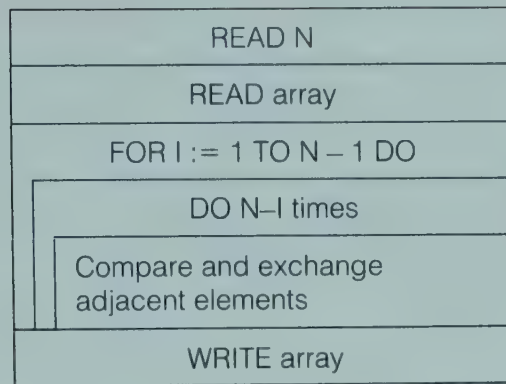


Figure 9-8

We have already written a piece of code to carry out the compare-and-exchange operation. Therefore, we probably need not construct a more detailed flowchart but will simply proceed to write the program.

```

PROGRAM BUBBLESORT ( INPUT, OUTPUT );
  ( * BY BUBBLESORT ALGORITHM,      * )
  ( * SORT N INTEGERS ( N <= 100 ) * )
  ( * READ INTO ARRAY A              * )
  VAR I, J, N, TEMP : INTEGER;
      A : ARRAY[1..100] OF INTEGER;
  BEGIN
    READ (N);
    FOR I := 1 TO N DO
      READ (A[I]);
    FOR I := 1 TO N - 1 DO
      FOR J := 1 TO N - I DO
        IF A[J] > A[J+1]
          THEN ( * EXCHANGE * )
            BEGIN
              TEMP := A[J];
              A[J] := A[J+1];
              A[J+1] := TEMP
            END;
      FOR I := 1 TO N DO
        WRITELN (A[I])
      END.

```

Now, if we look carefully at the bubblesort algorithm, we notice that there is a way to make the sorting operation sometimes more efficient. If, on any pass, no exchanges are made, the array is clearly already sorted, and proceeding to carry out further passes is a waste of time. What we can do to avoid this waste of time is to (1) set a value of a variable, sometimes referred to as a "flag," to tell us whether an exchange has or has not been made on the current pass; and (2) check the value of the flag before proceeding to execute another pass.

To implement this idea, since the number of passes executed will vary, we see that we need to use a WHILE (or REPEAT) statement instead of a FOR statement to control the outer loop. Here, then, is another version of BUBBLESORT that stops as soon as the array is sorted.

```

PROGRAM BETTERBUBBLE (INPUT, OUTPUT);
(* USING BUBBLESORT, SORT N INTEGERS (N<=100) IN
   ARRAY A *)
VAR I, J, N, TEMP: INTEGER;
    A: ARRAY[1..100] OF INTEGER;
    FLAG: BOOLEAN;
BEGIN
    READ (N);
    FOR I := 1 TO N DO
        READ (A[I]);
    FLAG := TRUE;
    I := 1;
    WHILE (I<=N-1) AND (FLAG = TRUE) BEGIN
        FLAG := FALSE;
        (* SET FLAG TO FALSE AT BEGINNING OF EACH
           PASS *)
        FOR J := 1 TO N-I DO
            IF A[J] > A[J+1]
                THEN (* EXCHANGE *)
                    BEGIN
                        TEMP := A[J];
                        A[J] := A[J+1];
                        A[J+1] := TEMP;
                        FLAG := TRUE
                    (* SET FLAG TO TRUE IF EXCHANGE IS
                       MADE *)
                    END;
            I := I+1
        END (* WHILE *);
    FOR I := 1 TO N DO
        WRITELN (A[I])
    END.

```

9-9 A PROGRAMMING EXAMPLE: LOAN PAYMENT SCHEDULE

When a bank makes a loan which will take several years to be paid off (e.g., a house mortgage), it is customary for the bank to provide a loan payment schedule, so the borrower can see what progress is being made in paying off the loan. In this section we'll see how easy it is to generate a payment schedule with a computer.

First, what information is usually given in such a table? We need the payment dates (month and year), the amount of the monthly payment (presumably fixed for the whole table—except possibly the last line), the amount of the payment which goes to pay the interest, the remainder of the payment (which is applied to reducing the principal), and finally the new balance owed. Columns of the table might then have headings:

MONTH	YEAR	INTEREST	PRINCIPAL	NEW BALANCE
-------	------	----------	-----------	-------------

An overall table heading would contain the original amount of the loan, the annual interest rate, and the monthly payment. These items (plus the starting date) are the input data that we will need for the computation.

Next, how is the desired information computed? Each month, the interest due is computed as one-twelfth of the annual interest rate times the balance owed (i.e., the new balance obtained after deducting the payment toward the principal in the previous month). And as noted above, the payment each month is just what's left over after the interest is paid. So that this is completely clear, let's look at an example. Suppose the original loan is for \$10,000, the annual interest rate is 12 percent, payments are \$190 a month, and they begin in June 1979.

The first interest payment due is then

$$\frac{0.12}{12} \times 10,000 = 100$$

We display the first two lines of the table, so you can see how it goes.

MONTH	YEAR	INTEREST	PRINCIPAL	NEW BALANCE
6	1979	100.00	90.00	9910.00
7	1979	99.10	90.90	9819.10

Now that we know how to do the computation by hand, how can we tell the computer how to do it? Clearly, we need a loop. Furthermore, we have no idea how many payments will be needed, so we must use either a REPEAT or a WHILE. Suppose we choose a WHILE. So far, this looks like any standard program that uses a loop to print successive lines of a table. However, there are two points of interest. First, the last line of the table will be a bit different. The regular monthly payment may be more than enough to pay off the loan, and this should be noted. Second, notice that we must somehow keep track of the changing months and years. Every 12 months, the year changes. You might think of running a 12-month loop to handle this, but, if you think about it, such a loop would mesh badly with the WHILE loop. So we propose to use an IF to test when the MONTH should be reset to 1 and the YEAR tick over. After this preliminary planning, we construct the flowchart shown in Figure 9-9.

Writing the following program is now quite easy.

```

PROGRAM SCHEDULE ( INPUT, OUTPUT );
VAR MONTH, YEAR : INTEGER;
    PAYMENT, BALANCE, RATE, PRINCIPAL, MONRATE,
    INTEREST : REAL;
BEGIN
    READ (MONTH, YEAR, PAYMENT, BALANCE, RATE);
    WRITELN ( '      LOAN PAYMENT SCHEDULE' );
    WRITELN;
    WRITELN ( ' LOAN:', BALANCE:11:2, '; RATE:',
        RATE:7:3, '; MONTHLY PAYMENT:', PAYMENT:9:2 );
    WRITELN;
    WRITELN ( ' DATE INTEREST PRINCIPAL BALANCE' );
    WRITELN ( ' *****' );
    MONRATE := RATE/12;
    INTEREST := MONRATE * BALANCE;
    WHILE (BALANCE + INTEREST > PAYMENT) AND
        (YEAR < 2050) DO
        BEGIN
            PRINCIPAL := PAYMENT - INTEREST;
            BALANCE := BALANCE - PRINCIPAL;
            WRITELN (MONTH:2, YEAR:5, INTEREST:10:2,
                PRINCIPAL:12:2, BALANCE:13:2);
            INTEREST := MONRATE * BALANCE;
            MONTH := MONTH + 1;
            IF MONTH > 12
            THEN BEGIN
                MONTH := 1;
                YEAR := YEAR + 1
            END (* END IF *)
        END; (* END WHILE *)
    WRITELN (MONTH:2, YEAR:5, INTEREST:9:2,
        BALANCE:9:2, ' **FINAL PAYMENT**')
END. (* SCHEDULE *)

```

Notice that we have put an extra condition in the WHILE loop to guard against an accidental infinite loop. Such a loop could be caused, for example, by running the program with a monthly payment less than the initial interest owed, so that the balance owed grows instead of decreasing. This particular mistake should really be guarded against by a validity check just before the WHILE loop:

```

IF PAYMENT <= INTEREST
THEN WRITELN ( 'PAYMENT <= INTEREST' )
ELSE (* WHILELOOP *)

```

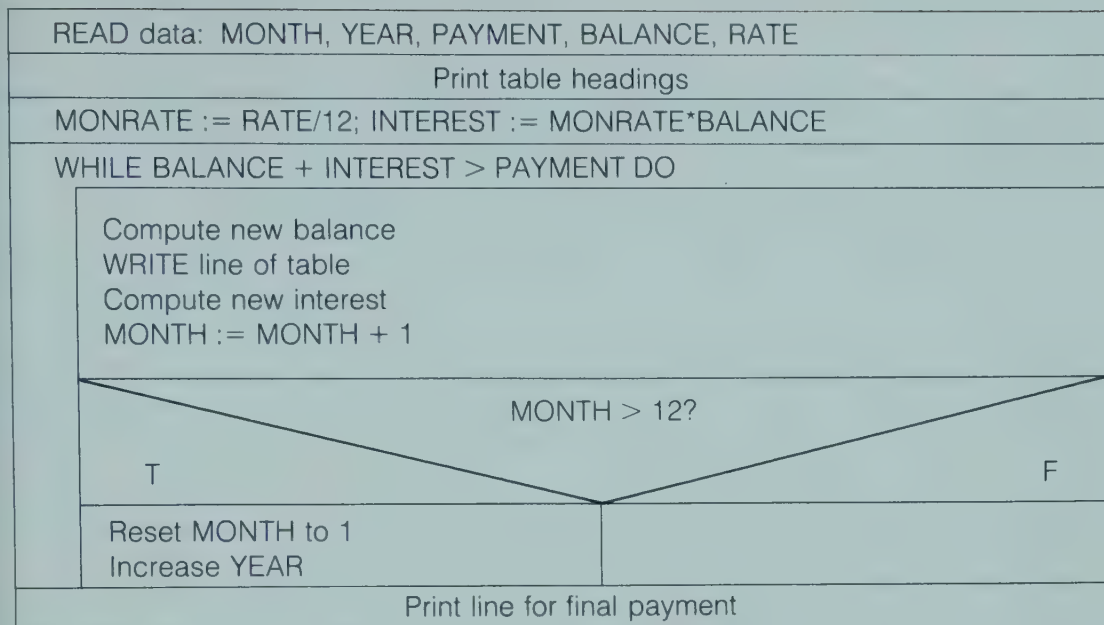



Figure 9-9

But the safeguard in the WHILE loop is still worth having, in case there's some fatal data error not caught by this check. Furthermore, if, say, PAYMENT = 190.00 and INTEREST = 189.99, we are unlikely to want to run the program to completion, even though the data passes the validity check.

Also, notice that we have inserted comments after the ENDS to indicate what is being ended. This is standard programming practice, particularly recommended for programs with nesting that might cause some confusion.

9-10 EXERCISES

9-1. Twenty-five integers are stored in a one-dimensional array A.

- (a) Write a program segment which replaces any occurrence of zero with the number 1.
- (b) Write a program segment which replaces the values 1, 2, and 3 with 2, 3, and 1, respectively, using a CASE statement.
- (c) Assume all the values in the array are 1, 2, or 3. Write a program segment which replaces 1, 2, and 3 with 2, 3, and 1, respectively, using neither an IF nor a CASE statement.

9-2. What is the output of the following?

```
BEGIN
  A := 5;
  B := 3;
  IF A < B
    THEN;
      WRITE ('PROBABLE SYNTAX MISTAKE');
  WRITELN (' IN PROGRAM')
END.
```

- 9-3.** Suppose you must write a program that examines a student record, looking to see whether the student:
- (a) Is on probation or in good standing (assume a BOOLEAN variable PROBATION that, when true, indicates that the student is on probation).
 - (b) Has just received a set of grades which (if PROBATION is true) flunks him out or (if PROBATION is false) puts him on probation. Assume REAL variables GRADE (which contains the student's average for the semester) and CUTPOINT (which contains the grade below which the student flunks out or is put on probation).
- Write an IF statement to write appropriate messages (and change the value of PROBATION, if necessary), depending on the values of PROBATION and GRADE.
- 9-4.** Write a program that reads a deck of cards (or successive lines of terminal input), each containing a pair of numbers A and B, and computes the sum S1 of those numbers B for which $A < B$, the sum S2 of those numbers B for which $A = B$, and the sum S3 of those numbers B for which $A > B$. Also print S1, S2, and S3. Assume that all values of A and B are positive and that the last card contains two zeros to indicate the end of the data.
- 9-5.** Suppose that the value to be assigned to a variable C depends on the signs of A and B as shown in the table below. Write a program segment to assign the correct value to C for any combination of values of A and B.

	A=0	A>0	A<0
B=0	C=1	C=2	C=3
B>0	C=2	C=1	C=5
B<0	C=4	C=6	C=1

- 9-6.** Which of the following statements will print the smallest of three distinct (i.e., different) numbers assigned to variables X, Y, and Z? If a statement fails, give an example of three numbers X, Y, and Z for which it does fail.

(a) IF X < Y THEN IF X < Z THEN WRITELN(X)
ELSE WRITELN(Z)
ELSE IF Y < Z THEN WRITELN(Y)
ELSE WRITELN(Z);

(b) IF (X < Y) AND (X < Z) THEN WRITELN(X)
ELSE IF Y < Z THEN WRITELN(Y)
ELSE WRITELN(Z);

(c) IF (X < Y) AND (Y < Z) THEN WRITELN(X);
IF (Y < Z) AND (Z < X) THEN WRITELN(Y);
IF (Z < X) AND (X < Y) THEN WRITELN(Z);

9-7. Are the following statements equivalent?

- (a) IF A < 5
 THEN IF B > 2
 THEN X := 7;
- (b) IF (A < 5) AND (B > 2) THEN X := 7;

Are the following conditions equivalent?

- (a) NOT ((A > 5) OR (B < 6))
- (b) (A <= 5) AND (B >= 6)

9-8. Notice that when a condition such as (X AND Y AND Z) occurs, if the expression X is false then, without evaluating Y and Z, you know that the whole condition is false. Some compilers take advantage of this fact and save some time by producing a machine language program that doesn't evaluate Y and Z when X is false. If your compiler does this, then a statement such as

IF (A <> 0) AND ((B/A) > 10.0) THEN ...

is valid; otherwise, when $A = 0$ you get an error from trying to divide by zero to evaluate the second expression. Run a test program to see what your compiler does. Give examples of other situations in which a partial evaluation of a boolean expression suffices to determine the value of the whole expression.

9-9. We have devised a new experiment, but we are not sure that our techniques are valid. In order to test this we would like to check our results against known laws to make sure we haven't violated them. (A certain amount of incorrectness in these results is allowable as experimental error.) In particular, we have measured some pressures, volumes, and temperatures and will use the law $PV = RT$ (the ideal gas law) to check our results. We have a set of $K + 1$ data cards. The first card has a value for R (the gas constant) and an integer K . The last K cards each contain a pressure, volume, and temperature reading from the experiment. (The units are compatible.) Our apparatus measures temperature in the range $100 \leq T \leq 500$ and pressure and volume so that their product is in the range $20 \leq P \cdot V \leq 5000$.

Write a program which reads in the data cards and:

- (a) Prints an appropriate message if it finds a data card containing values which have been obviously mispunched and ignores that card with respect to (b) and (c).
- (b) Computes the number of sets of data such that PV/T is within a factor of 0.06 of R . Call this value A.
- (c) Computes the number of sets of data such that PV/T is not within a factor of 0.60 of R . Call this value B.

Print the values of A and B. If A is less than 95 percent of K we want to revise the experiment. Print an appropriate message to indicate if revision is required.

- 9-10.** A percentage of sales is offered to each salesperson as a commission. The percentage depends upon the amount sold and whether the item is of class 1, 2, 3, or 4.

Class 1: For amount equal to or less than 1000, the commission is 6 percent.

For amount greater than 1000 but less than 2000, the commission is 7 percent.

For 2000 or greater, the commission is 10 percent.

Class 2: For amount less than 1000, the commission is 4 percent.

For amount 1000 or greater, the commission is 6 percent.

Class 3: For all amounts, the commission is $4\frac{1}{2}$ percent.

Class 4: For all amounts, the commission is 5 percent.

Write a Pascal program that reads in data cards describing sales and prints out the salesperson's commission. Each data card should contain **(a)** salesperson's number, **(b)** class of item sold, **(c)** unit price of item, and **(d)** amount sold. The end of the data is indicated by a card with four zeros. Output should be an appropriately headed table with columns:

Salesperson's number

Class of item sold

Unit price of item

Amount sold

Total sale in dollars

(amount * unit price)

Commission on sale

(in dollars)

- 9-11.** Relations on sets play an important role in the physical and social sciences. For example, suppose that we have a set of six people which we represent by $\{P_1, P_2, \dots, P_6\}$. And suppose that we take "is a sibling of" to be our relation. That is, P_i is related to P_j if, and only if, P_i is either the brother of or the sister of P_j . Suppose, in this example, that P_1, P_2 , and P_3 are three sisters in one family, that P_4 and P_5 are brother and sister in another family, and that P_6 is an only child. We could represent this sibling relation by a 6×6 matrix as follows: There is a 1 in column j of row i if P_i is a sibling of P_j ; there is a 0 in column j of row i if P_i is *not* a sibling of P_j . Thus, the matrix for this relation would be

0	1	1	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	0	0	0

In general, if we have a relation on a set with n elements, then we can represent that relation with an $n \times n$ matrix as follows:

The value in the i th row and j th column of the matrix is 1 if the i th element is related to the j th element, and 0 if the i th element is not related to the j th element.

Relations and their associated matrices can have various properties such as the four listed below.

Reflexive

A relation is reflexive if and only if (iff) every element of the set is related to itself. Example matrix:

1	0	1	1
0	1	1	0
1	0	1	0
0	0	0	1

Symmetric

A relation is symmetric iff for all elements a and b of the set, if a is related to b , then b is related to a . Example matrix:

1	0	1	1
0	0	1	0
1	1	1	0
1	0	0	0

Antisymmetric

A relation is antisymmetric iff for all elements a and b of the set, if a is related to b and b is related to a , then a must equal b . Example matrix:

0	1	0	1
0	1	0	0
1	0	0	0
0	1	0	1

Transitive

A relation is transitive iff for all elements a , b , and c of the set, if a is related to b and b is related to c , then a is related to c . Example matrix:

```

0  1  1  0
0  0  1  0
0  0  0  0
0  0  0  1

```

Write a program that reads in a matrix that represents some relation on an arbitrary set, prints out the matrix, and prints out a statement stating that the matrix does or does not have each of the aforementioned properties. Your program must do this for a number of matrices. Sample output for one matrix:

THE 4 × 4 MATRIX

```

1  1  1  0
0  1  1  0
0  0  1  0
0  0  0  1

```

```

IS REFLEXIVE
IS NOT SYMMETRIC
IS ANTISYMMETRIC
IS TRANSITIVE

```

Your input file will contain a sequence of $n \times n$ matrices. For each matrix, the size (n) is given first. This is followed by the matrix elements listed row by row (row major order). The end of the file is indicated by specifying size 0. A sample input file containing only the matrix given above is:

```

4
1  1  1  0
0  1  1  0
0  0  1  0
0  0  0  1
0

```

Your program should work for any matrix of dimension 2×2 up to 10×10 .

- 9-12.** A new type of vending machine is under development which will be controlled by a computer using a Pascal program. This program will accept any denomination of coin

(1 cent, 5 cents, 10 cents, 25 cents, or 50 cents). Correct change will be returned for any amount tendered for any price of an item. You are to write a Pascal program to compute the denomination and number of coins to be returned in change.

Each morning the machine is supplied with an appropriate number of coins of each denomination to be used in making change. Change is to be made only from these units, and not from any money received for purchase. Change is to be returned using the minimum number of coins, i.e., one half-dollar rather than two quarters, one dime rather than two nickels, etc. In the event that one unit of change is exhausted during the day, smaller units may be used. In the event that the machine is unable to make exact change, the purchase is to be refused. To simulate the operation of this machine with a Pascal program, proceed as follows:

- (a) Read the number of half-dollars, quarters, dimes, nickels, and pennies to simulate loading the machine with change. These values will be integers, less than 100, on a single data card, and in the order given above.
- (b) Print a single line giving the values just read in.
- (c) Begin the main program loop.
 - (1) You will read in two values: amount tendered and amount of sale. These will be integers indicating the price in cents. Both values will be less than 1000, and the amount of sale will be less than or equal to amount tendered.
 - (2) Print, all on a single line, the amount tendered, amount of sale, and change to be returned.
 - (3) Calculate the number of each denomination of coin to be returned as change.
 - (4) Print, on a single line, the coins to be returned in the order: half-dollars, quarters, dimes, nickels, and pennies. In the event that exact change cannot be returned, print the words NO SALE on this line instead.
 - (5) Repeat the loop until a value of zero is read for amount tendered. The card which contains this value will contain two zeros to indicate the end of the data.
- (d) After processing all sales, print the number of coins of each denomination remaining.

9-13 Convert Example 9-2 into an interactive program that prompts for N, converts it, and loops to request a new N. Suggested prompt message: "Enter a positive integer less than 5000 or enter 0 to stop."

SUMMARY OF CHAPTER 9

The IF statement enables us to branch out in one of two possible directions. If the value of the controlling boolean expression is TRUE, the task specified in the THEN branch will be executed. If the value of the controlling boolean expression is FALSE, then the task specified in the ELSE branch is executed.

Nested IF statements can be used for multiway branching. Alternatively, the CASE statement can be used. In a CASE statement, the current value of the case index determines the branch of execution to be followed.

10

CHARACTER VARIABLES AND CHARACTER STRINGS

10-1 INTRODUCTION

The modern digital computer was developed in the time of World War II to satisfy the need for long and complicated numerical computations related to the development of weaponry. For a number of years the major applications of computers continued to be scientific and engineering computations. But by now computer applications have broadened far beyond scientific numerical computations. For example, computers are used to handle payrolls, to keep track of personnel records, to control inventories, and to assist editorial and typesetting work for newspapers and magazines. All these applications involve the processing of text information: names, words, and other non-numeric data. Consequently, any modern programming language must provide facilities for handling such kinds of information.

You have already made extensive use of one such facility available in Pascal. This is the facility to print an arbitrary character string by a statement such as

```
WRITELN ( 'TABLE HEADING' )
```

Through using this facility, you have also gained a basic familiarity with what constitutes a "character string"—namely, that it is just a "string" of "characters" enclosed in single quotes. Furthermore, you have become familiar with the set of characters (including special symbols like *, <, >, etc.) available to you, and you have learned that a *blank*, or *space*, is just another character that can be included in the string.

It may have occurred to you that such character strings, in order to be written out, must be somehow stored in the computer's memory. And it may also have occurred to you that language facilities to read, store, compare, and manipulate such strings would be enormously useful. Consider programs to compute student grades. In the examples we gave, we identified students by number. Why not by name? That is, we'd like to be able to put a student's name on the grade card and then print total grades identified by name. And consider the CASE statement (Example 9-3) that printed the name of the Nth day of the week. We could handle this problem more neatly if we could fill an array named DAYNAME with character strings so that, for example, DAYNAME[1] has the value SUNDAY. Then that whole CASE statement could be replaced by something like

```
WRITELN (DAYNAME[N])
```

You have seen enough of Pascal by now to realize that what we are getting into here is a new *data type*—specifically, the character type. We will discuss character variables in the next section and then explain how, with the Pascal language, one fits characters together to form strings.

10-2 CHARACTER VARIABLES

So far we have discussed variables of three different types: INTEGER, REAL, and BOOLEAN. Here we introduce a fourth type: CHAR (short for character). In Pascal a CHAR constant is any printable character—a letter, a digit, a blank, or any of the special characters like /, *, ?, etc. It is represented by a quotation mark ('), followed by the character itself, followed by another quotation mark. A CHAR variable has a CHAR constant as its value. CHAR variables may be read, written, and used in conditions and assignment statements. Thus, for example, in the following program segment, the variable INITIAL is declared to be a CHAR-type variable and its value is set to be 'G'.

```
VAR INITIAL: CHAR;
BEGIN
    INITIAL := 'G';
```

Don't forget that a blank is considered a printable character. Its representation is ' ', of course. The quotation mark itself is a printable character, which, as we discussed in Chapter 2, is represented by '' '. Thus, the following Pascal segment sets the value of the CHAR variable BLANK to blank and the value of the CHAR variable QUOTE to the special character ' (the single quote).


```

VAR BLANK, QUOTE : CHAR;
BEGIN
    BLANK := ' ';
    QUOTE := ''';

```

Inside the computer each character is stored as a coded sequence of 0's and 1's—just like all the other information in the computer. (This sort of representation shouldn't be new to you. It's the same idea as the well-known Morse code representation of characters by sequences of dots and dashes.) The quotation marks which surround a character value in a Pascal program are not part of the representation in the computer. Their purpose is to distinguish the character X from a variable named X or the character 1 from the integer 1 in the Pascal language. Thus, if ONE is a CHAR variable and a program includes the statement

```
ONE := '1'
```

the encoding for the *character* 1 is stored as the value of ONE. But if ONE is declared an INTEGER variable and the code includes

```
ONE := 1
```

the representation of the integer 1—a *number*—is stored as the value of ONE.

The fact that characters appear in the computer as sequences of 0's and 1's clarifies the point that a blank is a character like any other. When we write

```
WRITE ( ' ' )
```

this causes the sequence of 0's and 1's that represent the blank to be sent to the printer. And when the printer sees the code for blank it spaces over one—the same effect you get by pushing the space bar on a typewriter.

There is another important consequence of representing characters by sequences of 0's and 1's. That is that the coded characters look just like numbers. We don't want to do arithmetic on these codes, but it is useful to do comparisons. In fact, we can compare any two characters using the relational operators `<`, `>`, `=`, `<=`, `>=`, or `=`. For example, we can write the conditions `'A' < 'K'` and `'A' <= 'K'`. As you will see, being able to compare letters in this way will allow us to write a program to put names, index entries, etc., in *alphabetical order*. You might wonder whether `'*' < 'A'`, or whether `'*' > '1'`. The ordering of all of the characters printable by a computer is called the *collating sequence* of that computer. Since different computer systems use different codes for characters, we can't tell you exactly what the sequence will be for your computer. However, there are two relationships that are

standard and yield the results for certain comparisons that we intuitively expect. These are:

1. The digits form a contiguous sequence:

$$'0' < '1' < '2' \dots < '9'$$

(By "contiguous" we mean that no nondigit characters are mixed in—i.e., we can't have $'1' < '+' < '2'$, for example.)

2. The letters form a contiguous sequence:

$$'A' < 'B' < 'C' < \dots < 'Y' < 'Z'$$

One rather important question regarding character order is whether the blank precedes or follows the letters. Here is an example program that makes use of a simple condition involving characters to determine the answer to this question. You might want to try it on your computer.

EXAMPLE 10-1

```
PROGRAM TEST ( INPUT , OUTPUT );
BEGIN
  IF ' ' < 'A'
    THEN
      WRITELN ( 'BLANK PRECEDES A' )
    ELSE
      WRITELN ( 'BLANK FOLLOWS A' )
END .
```

As you will see when we come to discuss strings of characters, the answer to this question will tell us whether or not the word CAR precedes CART, as it does in the dictionary.

Another interesting question is: Do the digits precede or follow the letters? (You can make up your own test for this one.)

Now here is an example of a complete program that uses character variables. This program reads an integer N, followed by a sequence of N characters. As each character is read, it is also written—except for the character T, which is replaced by an * in the output sequence.

EXAMPLE 10-2

```
PROGRAM REPLACE ( INPUT , OUTPUT );
VAR CH : CHAR;
    I , N : INTEGER;
BEGIN
  READLN ( N );
  FOR I := 1 TO N DO
```

```

                                BEGIN
                                READ (CH);
                                IF CH = 'T'
                                    THEN WRITE ('*')
                                    ELSE WRITE (CH)
                                END
END .

```

Notice that the first READ statement in this program is a READLN. This is so that we can put the integer N on a separate line of input (e.g., on a separate card). If we used READ instead of READLN, the characters read by the loop would be the first N characters after the number N, including blanks and any end-of-line marker. Thus if the characters to be read in were STRING SUBSTITUTION, the data would have to appear as

19STRING SUBSTITUTION

But by using READLN, we can put the count and the characters on separate lines:

19 STRING SUBSTITUTION

The READ statement in the loop will begin reading in the leftmost column of the second line. Not only any blanks between words but also any blanks at the left must, of course, be included in the count of characters to be read and printed. A lesson to be learned from this example is that reading characters requires considerable care. Reading numbers is much easier, since the computer just looks for the digit that begins the next number, skipping the blanks that separate numbers. Notice also that no quotes appear in the input data. Each character is transmitted from the input device to the computer in coded form and then is stored in the location named CH. The fact that this is character data is known to the computer because CH has been declared as a variable of type CHAR.

The statement WRITE (CH) writes a single character; namely, the value of CH. This is just the effect we want in this example. If we want to space the characters out, we can specify a fieldwidth, just as in printing numbers. Thus

```
WRITE (CH : 9)
```

prints eight blanks before the value of CH. One use of this feature is to print a string of blanks. For example,

```
WRITE (' ' : 20)
```

will write 20 blanks.

Let us complete this section by giving you an example of a useful program that handles characters. You've probably seen statistics on the frequency of occurrence of letters of the alphabet. (For example, "e" appears most frequently in ordinary English text.) This program will allow you to type a page of text into the computer and have it count the number of occurrences of each letter. (You could use it, for example, as a first step in decoding text that has been encoded by the method of substituting one letter for another, as in the popular cryptograms.) So that you don't have to count the number of characters, we'll assume the text contains no asterisks and an * is used to signal the end of the input. There are two interesting features that we wish to point out in this program. First, notice that to add the counts for each letter into an array, we've used a CASE statement with character values identifying the cases. Second, we had to nest the CASE statement inside an IF statement so that characters that are not letters are skipped over. (Remember that, depending on your system, it may be an error to try to execute a CASE statement for a value of the index that is not among those listed.)

EXAMPLE 10-3

```

PROGRAM COUNTTEXT (INPUT, OUTPUT);
VAR
    I: INTEGER;
    CH: CHAR;
    COUNTS: ARRAY[1..26] OF INTEGER;
BEGIN
    FOR I := 1 TO 26 DO
        COUNTS[I] := 0;
    REPEAT
        READ (CH);
        IF (CH >= 'A') AND (CH <= 'Z') THEN
            BEGIN
                CASE CH OF
                    'A' : I := 1;
                    'B' : I := 2;
                    'C' : I := 3;
                    'D' : I := 4;
                    'E' : I := 5;
                    'F' : I := 6;
                    'G' : I := 7;
                    'H' : I := 8;
                    'I' : I := 9;
                    'J' : I := 10;
                    'K' : I := 11;
                    'L' : I := 12;
                    'M' : I := 13;
                    'N' : I := 14;
                    'O' : I := 15;

```

```

        'P' : I := 16;
        'Q' : I := 17;
        'R' : I := 18;
        'S' : I := 19;
        'T' : I := 20;
        'U' : I := 21;
        'V' : I := 22;
        'W' : I := 23;
        'X' : I := 24;
        'Y' : I := 25;
        'Z' : I := 26;
    END; (* CASE *)
    COUNTS[I] := COUNTS[I] + 1
END (* THEN *)
UNTIL CH = '*';
FOR I := 1 TO 26 DO
    Writeln (I, COUNTS[I])
END.

```

10-3 PACKED ARRAYS

The introduction of character type variables enables us to handle text information. Since most character information is in the form of a string or sequence of characters (e.g., a name, an address, or a title of a book), we need to have facilities to handle such strings. A natural way to store a character string is to use an array of character variables, as the following example illustrates:

```

VAR NAME: ARRAY [1..40] OF CHAR;
BEGIN
    NAME [1] := 'J';
    NAME [2] := '.';
    NAME [3] := ' ';
    NAME [4] := 'H';
    NAME [5] := 'O';
    :

```

There can be, however, a serious disadvantage to using an array to store a character string: namely, inefficient use of memory locations. In the example above, we declare NAME to be an array with 40 elements. Consequently, 40 locations are reserved. In each location a single character will be stored. Such an arrangement is inefficient because, depending on the particular computer, a memory location can usually hold four to ten characters. In Pascal, such inefficient usage of memory locations can be avoided because there is another kind of array known as a *packed array*. In a packed array, the elements are stored packed as tightly together as possible. Thus, if in a

particular computer four characters will fit in one location, four elements of a packed array of characters are packed into one location. For example, we might have

```
VAR TITLE : PACKED ARRAY [1..40] OF CHAR;
```

If each location in our computer holds four characters, then the 40 elements of the array TITLE are packed into 10 locations. This is a dramatic saving of storage space. Figure 10-1 illustrates how the characters EXAMPLE are stored in an array (Figure 10-1a) and in a packed array (Figure 10-1b). Each memory location is assumed to hold four characters.

Packed arrays may be used just like ordinary arrays. For example, we can store a character value into one of the variables:

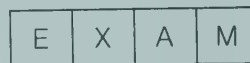
```
TITLE [1] := 'T'
```

or we can print out one of the elements:

```
WRITE (TITLE [3])
```



(a)



(b)

Figure 10-1

However, it can be quite time-consuming to manipulate individual elements of a packed array. This is because the computer not only must access the memory location holding the variable (as for an ordinary array) but also must do some extra work to extract only a portion of the information stored in that location.

The advantage of using packed arrays of characters lies not only in the saving in storage space but in the Pascal facility to treat such an array as a *unit*; i.e., to reference it by its name without a subscript and, for example, to store an entire character string into it by one assignment statement instead of letter by letter. We will discuss this facility in detail in the next section.

It should be noted that a BOOLEAN value does not occupy a whole memory location either, so we can also have packed arrays of BOOLEAN values:

```
VAR XRAY : PACKED ARRAY [1..20] OF BOOLEAN;
```

10-4 CHARACTER STRINGS

A character string is simply a string (or sequence) of printable characters. In many situations we want to be able to treat a character string as a single entity. Conceptually, we imagine a new data type called STRING. We want to define STRING constants and STRING variables; i.e., variables that have STRING constants as their values. We want to be able to READ and WRITE the values of STRING variables, to assign values to STRING variables, and to compare the values of STRING variables in conditions. In a later chapter, we shall see that Pascal allows us to actually define such a new data type; this will allow us to handle character strings very conveniently. But in this section we shall study how we can treat character strings as entities by using nothing more complicated than the data type PACKED ARRAY.

First, let us look at STRING constants. Such a constant may be defined in a constant declaration in the same way as any other constant. For example,

```
CONST  
    NAME1 = 'J. HO';  
    NAME2 = 'PARIS THEODORACOPULOS';
```

Once we have declared these constants, we can write, for example,

```
WRITELN (NAME2)
```

which is equivalent to

```
WRITELN ('PARIS THEODORACOPULOS')
```

As another example, suppose we have a string that we want to print many times—perhaps a string like

```
*****
```

that makes a nice separator between sections of output. We can define

```
CONST SEPARATOR = '*****';
```

Then we only need write

```
WRITELN (SEPARATOR)
```

whenever we want to print a separator line. If we want to print some blanks before the separator, we can use a fieldwidth specification. Thus

```
WRITELN (SEPARATOR : 20)
```

prints 10 blanks before the string of 10 asterisks.

Next, let us look more closely at variables that are defined as packed arrays of characters. Since the values they take on are strings of characters, these variables are frequently referred to as *string variables*. For example, we might declare a string variable STUDENT by

```
VAR STUDENT : PACKED ARRAY [1..10] OF CHAR;
```

Unlike ordinary arrays, we can reference the elements of a packed array collectively by the name given to it. Thus the declaration above not only sets up an array of 10 elements named STUDENT[1]...STUDENT[10], but also allows us to refer to the string variable thus defined as a single entity with the name STUDENT. As an example, consider the following program.

EXAMPLE 10-4

```
PROGRAM TESTSTRING (INPUT,OUTPUT);
VAR STUDENT : PACKED ARRAY[1..10] OF CHAR;
BEGIN
    STUDENT := 'JOHN SMITH';
    STUDENT[8] := 'Y';
    WRITELN (STUDENT)
END.
```

The result of this program is to print the line

```
JOHN SMYTH
```

Let us look closely at what has happened. We have assigned the character string 'JOHN SMITH' to the string variable STUDENT by an ordinary assignment statement. We then change the eighth letter in the string by assigning a new value to a single element of the array. Finally we print the value of the string variable, again referencing it as an entity.

To summarize, we see that we can treat a string variable as a single entity in assignment statements and in WRITE commands. Can we also READ values of string variables? The answer to this is No! If we insert the statement

```
READ (STUDENT)
```

into Example 10-4, we get the message that we have made a type error. The name of a packed array cannot be used in a READ command. In order to read a character string, we must proceed exactly as described in the previous section and read one character at a time. The easy way to read an entire string is to use a FOR loop. Thus, if we have declared the packed array STUDENT as above and wish to read in a value for it, we include in the program the statement

```
FOR I:=1 TO 10 DO  
  READ (STUDENT[I])
```

Before going on to discuss the comparison of character strings, let us note that the declaration of a packed array includes its *length*. An unfortunate feature of Pascal is that packed arrays of different lengths are considered to be of different *types*. For example, if you declare STUDENT to be a packed array of length 10 and then write

```
STUDENT := 'JOE SMITH'
```

the compiler will say you have made a type error. You can fix it by writing

```
STUDENT := ' JOE SMITH'
```

(i.e., putting an extra blank in the character string), but this is a nuisance. Similarly, if the name is too long for the variable, you will have to truncate it:

```
STUDENT := 'THEODORACO'
```

(Clearly, the strategy is to define a string variable such as STUDENT long enough to hold all possible names of students—but at the expense of having to type many blanks in strings for shorter names.) The impact of the fixed length of string variables is less severe if the value is to be read in, since the characters are input one at a time. If the input line has too many characters, the reading simply stops after the number of characters prescribed by the FOR loop, and any remaining characters are left unread. If the input does not

contain enough characters, and you are providing input by typing at a terminal, the program will just wait until you have provided enough characters. (If the value is typed on a data card, the program will just read the blanks or other characters following the value until enough characters have been read.)

Now let us look at how character strings may be compared, or, more precisely, how relational operators may be used with string variables and constants to form conditions. Such a facility allows us, for example, to search through a pack of data cards for information on Jones with a statement like

```
IF NAME = 'JONES' THEN ...
```

Or we can use a comparison

```
IF WORD2 > WORD1
  THEN ( * EXCHANGE * )
```

in a program to sort a list of words into alphabetic order. Notice that just as in WRITE and assignment statements, a string may be referenced as an entity in conditions. However, the fact that string variables of different lengths are of different types causes some difficulties in comparing strings. In short, strings of different lengths *cannot* be compared; attempting to do so will cause a type error. For example, suppose a program contains the following:

```
CONST NAME = 'JOE SMITH';
VAR STUDENT : PACKED ARRAY [1..10] OF CHAR;
    I : INTEGER;
BEGIN
  FOR I := 1 TO 10 DO
    READ (STUDENT[I]);
    IF STUDENT = NAME THEN
      .
      .
      .
```

The compiler will print a type-error message for this program since STUDENT is of length 10, while NAME is of length 9. Again, the only way to fix this problem is to add blanks appropriately and write

```
NAME = 'JOE SMITH '
```

Keeping this restriction clearly in mind, let us now look more closely at how strings of equal lengths are compared. The general procedure is to start at the left sides of the two strings, comparing character by character until a TRUE or FALSE conclusion is reached. Suppose, for example, the condition is

```
'CARRY' = 'CART '
```

The first three characters are compared and found to be equal. But comparing the fourth characters ('R' with 'T') yields an inequality and the condition is evaluated to FALSE. If the condition had been

'CARRY' <> 'CART'

it would have evaluated to TRUE as soon as the comparison of 'R' with 'T' was reached. In evaluating inequalities—conditions involving the operators <, >, <=, and >=—the rule is that the first nonequal characters determine the ordering of the strings. And the ordering of individual characters is, of course, that given by the collating sequence of the computer system, as we said in Section 10-2. Thus

'CARRY' < 'CART'

since 'R' < 'T'. We would also like to conclude that, as in the dictionary,

'CAR' < 'CART'

But since these two character strings are of different lengths we must write

'CAR ' < 'CART'

Whether this is true will depend upon the collating sequence for your computer, since the critical comparison is whether ' ' < 'T' is TRUE or FALSE.

To see how all this works, study the following complete program, which reads in two character strings and compares them, printing out whether they are equal or, if unequal, which comes first according to the collating sequence.

EXAMPLE 10-5

```
PROGRAM COMPARE(INPUT,OUTPUT);
VAR WORD1,WORD2:PACKED ARRAY [1..15] OF CHAR;
    I : INTEGER;
BEGIN
  FOR I := 1 TO 15 DO
    READ(WORD1[I]);
  WRITELN(WORD1); (* ECHO INPUT *)
  READLN; (* TO LET US FOLLOW WORD1 BY LOTS OF
    BLANKS AND BEGIN WORD2 ON A NEW LINE *)
  FOR I := 1 TO 15 DO
    READ(WORD2[I]);
  WRITELN(WORD2);
  IF WORD1 = WORD2 THEN
```

```

WRITELN ('STRINGS ARE EQUAL')
ELSE IF WORD1 < WORD2 THEN
    WRITELN ('WORD1 PRECEDES WORD2')
    ELSE WRITELN ('WORD1 FOLLOWS WORD2')
END.

```

To do some of the things that we have mentioned in this chapter (e.g., put a list of words into alphabetical order) we need to be able to talk about arrays of string variables—in other words, arrays of arrays. This is a somewhat complicated business in Pascal; indeed, it requires the facility to define a string variable as a new data type, as we mentioned at the beginning of this section. So we will delay providing you with elaborate examples of programs involving character strings until a later chapter. However, to provide you with one more simple example, here is a program that reads a name in one format and prints it out in another. The two formats are specified in the comment at the beginning of the program.

EXAMPLE 10-6

```

PROGRAM INVERTNAME (INPUT,OUTPUT);
    (* READ NAME IN FORMAT                                *)
    (*          SMITH, JOE T.                              *)
    (* WITH ANY NUMBER OF BLANKS BEFORE                   *)
    (* AND BETWEEN COMPONENTS.                             *)
    (* PRINT NAME IN FORMAT                                *)
    (*          JOE T. SMITH                                *)
VAR LASTNAME: PACKED ARRAY[1..20] OF CHAR;
    FIRSTNAME: PACKED ARRAY [1..10] OF CHAR;
    CH, INITIAL : CHAR;
    I ,N : INTEGER;
BEGIN
    REPEAT
        READ (CH)
    UNTIL CH <> ' ';
    I := 1;
    WHILE CH <> ',' DO
        BEGIN
            LASTNAME [I] := CH;
            READ (CH);
            I := I + 1
        END;
    REPEAT
        READ (CH)
    UNTIL CH <> ' ';
    I := 1;
    WHILE CH <> ' ' DO

```



```

      BEGIN
        FIRSTNAME[I] := CH;
        READ (CH);
        I := I + 1
      END;
    REPEAT
      READ (CH)
    UNTIL CH <> ' ';
    INITIAL := CH;
    (* NAME READ IN; NOW PRINT *)
    N := I - 1;  (* N = LENGTH OF FIRSTNAME *)
    FOR I := 1 TO N DO
      WRITE (FIRSTNAME[I]);
    WRITE (' ', INITIAL, ' ', LASTNAME)
  END.

```

Some points to note in this program are the following:

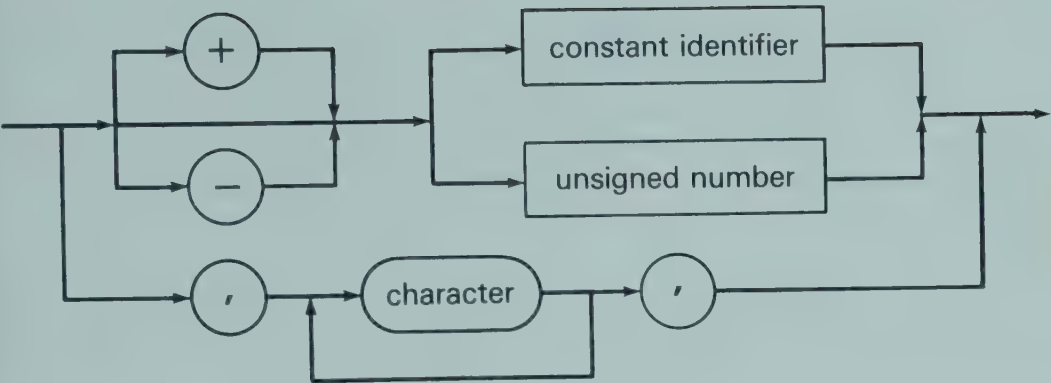
1. Study the REPEAT loops used to skip over strings of blanks. It is essentially this technique that is used when the computer reads lists of numbers. That is, the columns are read one by one until the computer sees a nonblank character. If the character is not a digit, you get an error message, like DIGIT EXPECTED, and the reading stops. (This is why numbers *must* begin with a digit, as 0.12.)
2. Different techniques are used to print FIRSTNAME and LASTNAME. Blanks at the end of LASTNAME don't matter, so we can simply WRITE the whole array. But to avoid the printing of extra blanks after FIRSTNAME, we use the character count automatically generated in the subscript I and print the characters of FIRSTNAME one at a time.

As an exercise, consider for what variations in input format the program should still work. For example, what if the typist puts a blank between the last name and the comma? Or no blank after the comma? What if the first name has a blank in it (e.g., ANN MARIE)? What if the last name contains a blank? What if the first name has more than 10 characters? What if the last name has more than 20?

10-5 SYNTAX

There is little that is new in the way of formal syntax in this chapter. The declaration of CHAR variables is like any other variable declaration. The addition of the description PACKED to an array declaration requires only a minor change in the diagram given in Section 7-6.

However, at this point we can provide you with the complete syntax diagram for *constant*, a concept which we have heretofore left as intuitive.



Thus we see that in Pascal there are three kinds of constants: those specified by identifiers (e.g., a declared constant such as the PI of Example 3-2 or one of the three standard constants TRUE, FALSE, and MAXINT); numbers (both REAL and INTEGER, with or without a sign); and character strings.

10-6 EXERCISES

10-1. Determine the output of the following program:

```
PROGRAM COUNT(INPUT,OUTPUT);
VAR WORD : PACKED ARRAY [1..4] OF CHAR;
    I, COUNT1,COUNT2 : INTEGER;
BEGIN
    COUNT1 := 0;
    COUNT2 := 0;
    FOR I := 1 TO 4 DO
        READ(WORD[I]);
        WHILE(WORD <> 'STOP') DO
            BEGIN
                IF (WORD >= 'IAAA') AND (WORD < 'NAAA')
                THEN COUNT1 := COUNT1 + 1
                ELSE COUNT2 := COUNT2 + 1;
                FOR I := 1 TO 4 DO
                    READ(WORD[I])
                END;
                WRITELN(' COUNT1 = ', COUNT1 : 4,
                    ' AND COUNT2 = ', COUNT2 : 4)
            END
        END
    END.
```

Use the data: ABE LYNNPETEMARK SUZYSTOP

10-2. Read in a character string of length less than 200. The end of the string is marked by \$. Delete all blanks from the string and print the compressed string. (Do not print the \$.)

- 10-3.** Extend the program COUNTTEXT (Example 10-3) in the following way. After filling the array COUNTS, compute the total number of letters and convert the count for each letter to a percent. The output should be the percent occurrence of each letter. Try out your program on a short paragraph. (Remember, you can let the input data run over several lines, or cards, since end-of-line markers are ignored like any other nonalphabetic character.)
- 10-4.** Write a program which reads in a character string of length less than 200. Check to see if the string is a palindrome, that is, a string which reads the same forward and backward if blanks and punctuation are ignored. (Assume that the input string ends with the character \$, but \$ is not part of the input to be examined.)
- 10-5.** Read in a string of text of length less than 500. Assume that the end of the text is marked by a \$. Print the following values:
- The number of sentences (ending with ., ?, !)
 - The average number of words in a sentence
 - The average number of letters in a word
- 10-6.** Consider the following algorithm for coding a secret message.
- For each letter of the message choose any word of length 3 or more, with the desired letter in the second position.
 - For each blank of the message choose any word of length 2.
 - Send this sequence of words followed by a \$.
- (For example, GO HOME could be sent as EGG YOLKS AT CHOSEN HOUR EMIT BENZENES.)
- Write a program which reads a coded message (one character at a time) and prints the decoded form.
- 10-7.** Bert and Ernie go bowling. But neither knows how to keep score. So they merely record how many pins fall for each ball bowled . . . and leave the score tabulation up to you.

A game of bowling is divided into 10 frames. Each bowler takes one turn per frame, with the same bowler taking his turn first in all frames of a given game. A turn for a bowler consists of either bowling two balls or knocking over 10 pins, whichever comes first. The score earned by a bowler for any frame is calculated as follows:

- If all 10 pins are knocked over with the first (thus, *only*) ball bowled, this is called a "strike" (denoted "X"), and the bowler's score for the frame is 10, plus the number of pins knocked over by that bowler's *next two* balls.
- If all 10 pins are knocked over, but using both balls, this is called a "spare" (denoted "-"), and the bowler's score for the frame is 10, plus the number of pins knocked over by that bowler with his *next* ball.
- The tenth frame is a special case:
 - A bowler receives *two* extra balls if he bowls a *strike* in the tenth frame. His final score is the total score through the ninth frame, plus 10, plus the number of pins knocked down by both of the two extra balls.

- b. A bowler receives *one* extra ball if he bowls a *spare* in the tenth frame. His final score is the total score through the ninth frame, plus 10, plus the number of pins knocked down by the extra ball.
- c. If a bowler bowls *neither* a strike *nor* a spare in the tenth frame, his final score is the total score through the ninth frame, plus the sum of the number of pins knocked down by the two balls bowled in the tenth frame.

This is a best-of-three contest between good friends, with Bert bowling first in each game. For each game, compute the score for each frame, the total score to that point in the game, and the total score for the game—for each bowler. Print the computed scores for each game in a legible horizontal format like that used in bowling. (See Figure 10-2.) You are expected to produce a similar output for each game bowled. Then print a message congratulating the winner of the contest. (Should one bowler win the first two games, do not calculate the score for the third game.) Bert and Ernie tied this time. The input for this example was:

5 4 6 3 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 4 6 10 5 4

	1	2	3	4	5	6	7	8	9	10			
BERT	5	4	X		X		X		X		X	4	6
	9	39	69	99	129	159	189	219	243			263	

	1	2	3	4	5	6	7	8	9	10			
ERNIE	6	3	X		X		X		X		X	5	4
	9	39	69	99	129	159	189	219	244			263	

Figure 10-2

- 10-8.** Write a program to translate a paragraph of text from English to pig Latin. The end of the input is marked by the special symbol \$, which is assumed not to occur in the text. Any English word can be translated into pig Latin by the following algorithm:

If the word does not start with a vowel, then move every letter up to but not including the first vowel to the end and add AY. Thus, CHAIR becomes AIRCHAY, and COMPUTE becomes OMPUTECAAY. If the first letter is a vowel, then add WAY to the word. Thus ART becomes ARTWAY. (Vowels are A, E, I, O, U, and Y, except that U is not a vowel if it comes after Q, and Y is not a vowel if it is the first letter and is followed by A, E, I, O, or U.)

As you translate words, you must also handle the punctuation marks: period, semicolon, colon, question mark, and comma. If one of these marks follows a word, then it must also follow the translation of the word. Thus TO; must be translated OTAY;.

- 10-9.** Write an interactive program similar to Exercise 8-11. Only now that you know how to handle character strings, read and print the employees' names as well as their identification numbers. Also, change the message asking whether there is more data to:

FINISHED? ENTER "YES" OR "NO"

SUMMARY OF CHAPTER 10

We introduce constants and variables of type CHAR (for character). A CHAR constant is any printable character. In a Pascal program, a CHAR constant is represented by a quotation mark, followed by the character itself, followed by another quotation mark. A CHAR variable has a CHAR constant as its value.

A string of characters can be stored conveniently in an array. The whole string can then be referred to by the array name (without subscript), while individual characters in the string are referred to as subscripted variables. A character string must be read one character at a time; that is, `READ (string)` is illegal.

To save space, a character string may be stored in a PACKED ARRAY, although it is less efficient to work with individual characters in a string when the array is packed. A string stored in a packed array may be easily written (with `WRITE (string)`) and may be compared with other strings.

In Pascal, strings of different lengths cannot be compared. Two strings of the same length will be compared on a character-by-character basis from left to right, according to the collating sequence of your computer system and compiler.



PROCEDURES AND FUNCTIONS

11-1 INTRODUCTION

Consider the following steps that the family doctor goes through to examine a patient:

1. Review the patient's medical history.
2. Measure the patient's weight and blood pressure.
3. Listen to the patient's heart and lungs.
4. Send the patient to a radiologist for a chest x-ray.
5. Study the report from the radiologist.
6. Send the patient to the laboratory for a blood test.
7. Study the report from the laboratory.
8. Send the patient to see a cardiologist.
9. Study the report from the cardiologist.
10. Send the patient to the laboratory for a urine test.
11. Study the report from the laboratory.
12. Send the patient back to the radiologist for more chest x-rays.
13. Study the report from the radiologist.
14. Send the patient to the laboratory for another blood test.
15. Study the report from the laboratory.
16. Declare the patient to be completely healthy.
17. Send the patient the bill.

Although this recital may be a little unreal, we do wish to use it to illustrate several concepts which are relevant to computing. We note that:

1. The family doctor does not perform every step of the examination herself. Rather, she seeks the assistance of the radiologist, the laboratory technician, and the cardiologist.
2. The radiologist, the technician, and the cardiologist provide assistance in their areas of specialty.
3. The radiologist, the lab technician, and the cardiologist can be called upon as many times as the family doctor deems necessary. They can be called upon to carry out different tasks at different times (e.g., to carry out a blood test and to carry out a urine test). They can also be called upon to carry out the same task at different times (e.g., to carry out two blood tests).
4. The examination begins and ends with the family doctor. Furthermore, she controls the sequence of steps that are carried out during the examination.
5. In general, when the patient is sent to a specialist for consultation, the family doctor will send along medical information that will be useful to the specialist and she will expect a report back from the specialist.
6. The specialists do not initiate any action on their own. They carry out prescribed procedures when called upon by the family doctor.

We can quickly see the analogy: A large and complex computational task might contain many subtasks which can be singled out in the same way the specialists' activities are separated from one another and from those of the family doctor. Some of these subtasks might be executed only once, while others might be executed a number of times. For example, in a program that processes the grades of the students in a class, we might wish to pick out the students with the highest grade in the first hourly examination. Later in the program, we might wish to pick out the students with the highest grade in the second hour examination, and then those in the final examination. Furthermore, we might also want to pick out those among all seniors, all juniors, all sophomores, and all freshmen who have the highest grade. It is not difficult to write a program segment to select the highest grade out of a set of grades. However, copying this program segment everywhere we need to select students with the highest grade will be unnecessarily wasteful. Our time is wasted in typing the same sequence of statements over and over, and computer memory space is wasted in storing the same sequence of instructions over and over. What we would like to do is to write a program segment that, being presented with a set of numbers, will pick out the largest one for us. Such a program segment would be analogous to the radiologist or cardiologist in the example of the medical examination of a patient. When the family doctor needs the help of the radiologist, she can simply send the patient to the radiologist and expect a report from him. By the same token, whenever we need to pick out the largest grade from a set of grades, we would call upon a specific program segment that will do the job for us. We would

supply this segment of program with the set of grades, and this segment of program would return to us the highest grade in the set.

In Pascal, there are two types of program segments that can be called on to perform special computational subtasks. They are called *procedures* and *functions*. We will discuss procedures first, and then indicate how functions are different. Because of the similarities, we will sometimes refer to both procedures and functions as *subprograms*.

In general, a program might contain many procedures. In a program, there is one and only one portion of the program that controls the sequence of steps of the computation, corresponding to the family doctor in our medical example. This portion of the program is known as the *main program*. Let us outline the situation, listing the main points just as in the medical examination analogy.

1. The main program does some but not necessarily all of the computation. It calls on procedures for assistance.
2. Each procedure carries out a special computational subtask.
3. Each procedure can be called on by the main program as many times as the main program desires.
4. Computation begins and ends with the main program.
5. In general, when the main program calls on a procedure, the main program should supply the needed data, and the procedure will return to the main program with the desired results.
6. A procedure does not initiate any computation on its own. Indeed, a procedure is totally inactive until called on by the main program.

Even if each procedure is to be called on only once, there is a great advantage to designing a program as a set of procedures which carry out independent tasks. (This is called *modular program design*.) Namely, it makes the program as a whole easier to write and to understand. The main program can consist of a sequence of calls to procedures to carry out their tasks, so we can immediately see from the main program what the successive tasks are. To return to our analogy, consider how complicated it would be to grasp the overall structure of a physical exam if for each step we included a complete list of what happens—e.g., how the x-ray is taken, developed, studied, etc.

We are now ready for the technical details.

11-2 DEFINING AND CALLING A PROCEDURE

As we pointed out above, a segment of a program that we can call on one or more times is called a procedure. A procedure is identified by a heading containing the word `PROCEDURE` and the name of the procedure. A procedure must have a name so that it can be referred to when the main program or other procedures want to call upon it to perform its task. A procedure name is

just an identifier, whose formation follows the general set of rules for identifiers. The computational task to be performed by the procedure is delineated by BEGIN and END. A procedure may also contain declarations of variables and constants after the header. In short, the body of a procedure, like the body of a program, is just a block. For example, we might define a procedure named HEAD as follows:

```
PROCEDURE HEAD ;
  BEGIN
    WRITELN( '*****' );
    WRITELN( 'ANNUAL PROGRESS REPORT' );
    WRITELN( '*****' );
  END;
```

The procedure HEAD, when called upon, will print out the heading of an annual progress report.

Once we have defined a procedure, we can call upon it as often as we wish from the main program or from another procedure. (On the other hand, although clearly wasteful, it is perfectly legal to define a procedure but never use it. The analogy is to have a radiologist in residence but never consult him for assistance.) To call upon a procedure from the main program, we simply mention the name of the procedure. Thus, if the following segment occurs in the main program

```
WRITELN ( 'FIRST TIME' );
HEAD ;
WRITELN ( 'SECOND TIME' );
HEAD ;
WRITELN ( 'NO MORE' )
```

the following lines will be printed:

```
FIRST TIME
*****
ANNUAL PROGRESS REPORT
*****
SECOND TIME
*****
ANNUAL PROGRESS REPORT
*****
NO MORE
```

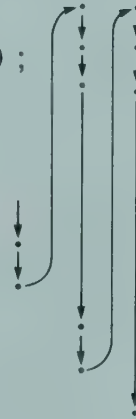
In particular, when the first statement of the main program segment is executed, the line FIRST TIME is printed. When the second statement is executed, PROCEDURE HEAD is called upon to print out the heading. The third statement is then executed and the line SECOND TIME is printed. PROCEDURE HEAD is then called upon again and the heading is printed. The last statement is then executed and the line NO MORE is printed.

To make sure that you understand how the program and the procedure work together, we shall put them together as a complete program and ask you to follow the arrows at the right, which indicate the sequence of steps to be carried out.

```

PROGRAM EX ( INPUT, OUTPUT );
  PROCEDURE HEAD;
    BEGIN
      WRITELN ( '*****' );
      WRITELN ( 'ANNUAL PROGRESS REPORT' );
      WRITELN ( '*****' );
    END;
  ( * BEGIN MAIN PROGRAM * )
  BEGIN
    WRITELN ( 'FIRST TIME' );
    HEAD;
    WRITELN ( 'SECOND TIME' );
    HEAD;
    WRITELN ( 'NO MORE' )
  END.

```



Notice that the procedure must be placed in front of the main program. This is necessary since otherwise the compiler will not understand the references to HEAD in the main program. In Pascal, there is a way to tell the compiler about procedures that will follow later, but this is a feature that is really needed only in certain special circumstances. We will therefore postpone discussion of this feature until Chapter 12.

11-3 VALUE PARAMETERS

Consider the following procedure.

```

PROCEDURE HEAD;
VAR I : INTEGER;
  BEGIN
    WRITELN ( 'ANNUAL PROGRESS REPORT' );
    FOR I := 1 TO 4 DO
      WRITELN ( '*****' )
  END;

```

Notice that since this procedure uses the variable I, I is declared within the procedure. Whenever PROCEDURE HEAD is called, it will print out the line ANNUAL PROGRESS REPORT followed by four lines of stars. Suppose that, under different circumstances, we wish to print the line ANNUAL PROGRESS REPORT followed by three lines of stars, or seven lines of stars, or eight

lines of stars. Instead of defining a different procedure to meet each of these requirements, we can define a procedure that will print out as many lines of stars as we wish. The number of lines to be printed will then be a *parameter* that we must specify each time we call the procedure. The parameters of a procedure are listed in the procedure heading, as follows.

```
PROCEDURE HEADX (N: INTEGER);
VAR I : INTEGER;
BEGIN
    WRITELN ('ANNUAL PROGRESS REPORT');
    FOR I := 1 TO N DO
        WRITELN ('*****')
    END;
```

In this heading we declare that PROCEDURE HEADX expects to receive the value of the parameter N when it is called. Furthermore, N should have an integer value. Clearly, if the value of N is specified to be 7, then seven lines of stars will be printed. Consider the following main program that calls PROCEDURE HEADX:

```
BEGIN
    WRITELN ('FIRST TIME');
    HEADX (7);
    WRITELN ('SECOND TIME');
    K := 3;
    HEADX (K);
    WRITELN ('NO MORE')
END.
```

(The variable K must of course have been declared in the main program.)
The statement

```
HEADX (7)
```

invokes PROCEDURE HEADX and specifies the value of N to be 7, and the statement

```
HEADX (K)
```

invokes PROCEDURE HEADX and specifies the value of N to be that of K (which is equal to 3). Of course, it is only for purposes of illustration that we use the statements

```
K := 3;
HEADX (K)
```

which have exactly the same effect as the statement

HEADX (3)

In a more realistic situation, the value of K might be determined through many complicated steps of computation.

A procedure can have as many parameters as we wish. Consider the following example.

```
PROCEDURE HEADXX (M, N: INTEGER);  
VAR I : INTEGER;  
BEGIN  
    FOR I := 1 TO M DO  
        WRITELN ( '*****' );  
    WRITELN ( 'ANNUAL PROGRESS REPORT' );  
    FOR I := 1 TO N DO  
        WRITELN ( '*****' )  
END;
```

To call PROCEDURE HEADXX, the values of the parameters M and N must both be specified. For example, the following statement

HEADXX (3 , 7)

will call HEADXX, specifying that the value of M is 3 and the value of N is 7. As you have probably already foreseen, the rule is that the values of the parameters should be given in the same order that the parameters are listed in the procedure heading. Thus, the two statements

```
HEADXX ( 3 , 7 )  
HEADXX ( 7 , 3 )
```

are not equivalent. What is printed out when each is executed?

Before going on to the next section, let us summarize some important facts about parameters and introduce some terminology for future use. The parameters used in a procedure are listed in two places. First, they are listed in the procedure heading. These are called *formal parameters*; they are just names (like the x you use in an algebraic formula) that must be given values when the procedure is invoked. Second, the parameters are listed in the procedure statement that invokes the procedure. These are called *actual parameters*; they tell the computer the actual values to be assigned to the corresponding formal parameters when the procedure is carried out.

The formal parameters and the actual parameters must match; i.e., there must be the same number of each, and they must be of corresponding types. Thus, since the formal parameters of HEADXX are declared to be INTEGERS, the actual parameters must also be of type INTEGER.

Parameters may be of type BOOLEAN or type CHAR as well as of type INTEGER or REAL. If you want a procedure to process a whole array, there is a problem, since it is *illegal* to include an array declaration in a procedure heading as follows:

```
PROCEDURE EXA (A : ARRAY [1..20] OF CHAR);
```

As the syntax diagram indicates (see Section 11-9), what follows the colon must be a type *identifier* (hence a single word). We will see in Chapter 14 how to get around this restriction.

11-4 VARIABLE PARAMETERS

Suppose we have a robot that makes dresses. Every time we push a button the robot will make a dress. So that we can turn out dresses of different sizes and styles, we would want to have a robot that will make a dress according to a pattern we supply every time we instruct the robot to make a dress. The analogy between dressmaking robots and procedures is clear. A procedure that has no parameter is like a robot that will make a dress of the same size and style each time its button is pushed. On the other hand, a procedure that accepts parameters is like a robot that will use the pattern we supply. Indeed, the pattern is like a parameter.

Now what about the material for the dresses? Presumably, the robot has a large supply of some one kind of material from which it cuts all the dresses. We might like to make the robot more flexible—so flexible that when we call on the robot to make a dress for us, we supply the robot with both a pattern and a piece of material. The robot will then use the material to make a dress according to the pattern. We can then think of the robot as being like a procedure with two parameters. We note, however, that there is a critical difference between the two “parameters” that we supply to the dressmaking robot. The pattern we supply to the robot is not to be destroyed. We might want to use it for other dresses. The robot is therefore designed to make a copy of the pattern and otherwise leave the pattern alone. Then it has complete freedom to do whatever it wishes to the copied pattern, without danger of destroying the original. On the other hand, the material we supply the robot is to be cut up and sewed together into a dress. In other words, the original material is to be transformed by the robot.

Correspondingly, there are two kinds of parameters we can supply to a procedure: those whose values will be copied but not altered and those whose values will be altered. The former are known as *value parameters* and the latter are known as *variable parameters*. In Section 11-3, we introduced the usage of value parameters. Here we shall discuss variable parameters.

Consider the following procedure named CHECK, which will add 100.00 to the value of CREDIT if it is larger than or equal to 500.00 and will change CREDIT to 500.00 otherwise.

```
PROCEDURE CHECK (VAR CREDIT:REAL);  
BEGIN
```

```
    IF CREDIT < 500.00
      THEN CREDIT := 500.00
      ELSE CREDIT := CREDIT + 100.00
END;
```

In the heading of the procedure, the word VAR signifies that CREDIT is a variable parameter, so the actual parameter will be altered. Thus consider the following segment of the main program:

```
    X := 275.00
    WRITELN (X:6:2);
    CHECK (X);
    WRITELN (X:6:2);
    CHECK (X);
    WRITELN (X:6:2)
```

It causes the following lines to be printed

```
    275.00
    500.00
    600.00
```

Let us illustrate the difference between value parameters and variable parameters by showing what will happen if in the procedure we inadvertently declare CREDIT to be a value parameter. This is easy to do; we just have to forget the VAR and write:

```
PROCEDURE CHECK1 (CREDIT:REAL);
BEGIN
  IF CREDIT < 500.00
    THEN CREDIT := 500.00
    ELSE CREDIT := CREDIT + 100.00
END;
```

In this case, the statements

```
    X := 275.00;
    WRITELN (X:6:2);
    CHECK1 (X);
    WRITELN (X:6:2);
    CHECK1 (X);
    WRITELN (X:6:2)
```

will produce the output

```
    275.00
    275.00
    275.00
```

What happens here is that when PROCEDURE CHECK1 is invoked, a copy of the value of X is made and identified by the name CREDIT. That is, there are actually two memory locations: one named X that is used by the main program and one named CREDIT that is used by the procedure. The value of X is copied into the location named CREDIT. The value of CREDIT is modified by the procedure, but there is no effect on X. On the other hand, PROCEDURE CHECK, where CREDIT was declared to be a variable parameter, when invoked by the statement

CHECK (X)

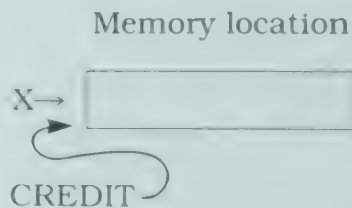
will modify directly the value of X. Indeed, in this case there is only *one* memory location involved. It is named X in the main program and CREDIT in the procedure. So naturally any changes in CREDIT by the procedure will be reflected in the main program as changes in X. To summarize the difference between these two types of parameters graphically:

Value Parameter: CREDIT



Number in memory location is copied when procedure is invoked.

Variable Parameter: CREDIT



CREDIT refers to same location as X.

In general, a procedure might have both value parameters and variable parameters. For example,

PROCEDURE EXAM (VAR X, Y: REAL; A, B: REAL)

This is a quite common situation. A procedure generally carries out some computation on numbers provided to it by the main program. It is natural for these numbers to be value parameters. Then the results of the computation must be handed back to the main program. This must be done via variable parameters, since the main program cannot "see" any changes made by the procedure to the values of value parameters.

11-5 SCOPE OF VARIABLES: LOCAL AND GLOBAL VARIABLES

One question you might be asking yourself is whether it is legal to use the same variable (or parameter) name in the main program and in a procedure—and if it is legal, whether the name refers to the same memory location or not. Indeed, the same name can be used in the main program and again in one or more procedures. Or the same name may be used in different procedures. The question of whether the name always refers to the same memory location has a rather complicated answer, as you will see.

The idea of having names that mean different things in different contexts should be familiar to you. If you live in Illinois and a friend tells you that he is going to Paris (or maybe Cairo or Vienna) for the weekend, you are not surprised. You assume that he means the town in Illinois by that name, and not the city in France (or Egypt or Austria). The names Paris, Cairo, and Vienna are *locally* (i.e., within Illinois) known to refer to towns within Illinois. On the other hand, if you live in New York and a friend tells you that he is going to Paris, you probably would be quite envious. You assume that he means the globally famous capital of France. Thus Paris is known in Illinois to mean one city and outside Illinois to mean an entirely different city. But residents of Illinois do not always have a provincial view of things. If your Illinois friend tells you he is going to London next week, you assume he's talking about the city in England, since there is no town named London in Illinois and London is globally known as the name of a city in England.

Analogously, within Pascal programs there are both locally known and globally known names. A global name is one that is known to the main program. Global names include identifiers and constants declared for the main program; i.e., they appear in the declarations under the PROGRAM heading. The names of procedures called by the main program are also global names. A local name is one that is known only to a particular procedure. Local names include the variables and constants declared under the PROCEDURE heading. Thus, if LONDON is a global variable (declared in the main program), and no procedure declares LONDON as a (local) variable, then any reference to LONDON will mean the global LONDON. But suppose a global variable PARIS has been declared in the main program and a certain procedure also declares a variable named PARIS. Then any reference to PARIS within the procedure means the local variable, while any reference to PARIS in the main program means the global variable. The two variables named PARIS are completely unconnected and so refer to completely different memory locations. You can think of it as simply a coincidence that the main program and the procedure use the same variable name for two different memory locations.

There is one important way in which our “real-world” analogy does not extend to Pascal programs. If your friend says he is going to Paris for spring vacation, he might mean the city in France. You would probably ask him which Paris he meant. In a computer language, such ambiguity is not allowed. If a procedure has a variable named PARIS, all references to PARIS within that procedure are to the local PARIS. It's as if the residents of Illinois,

having named a town Paris, were forever prevented from talking about—or visiting—Paris, France.

Here's a simple example to clarify all these ideas.

EXAMPLE 11-1

```

PROGRAM TRIV1 (INPUT, OUTPUT);
VAR K, I, TOTAL: INTEGER;
    PROCEDURE SUM (VAR N, S: INTEGER);
        VAR I: INTEGER;
        BEGIN
            S := 0;
            FOR I := 1 TO N DO
                S := S + I
            END;
        END;

BEGIN (* TRIV1 *)
    READ (K);
    FOR I := 1 TO K DO
        BEGIN
            SUM (I, TOTAL);
            WRITELN (I, TOTAL)
        END
    END.

```

In this trivial example, PROCEDURE SUM computes the sum of integers from 1 to N. There are two variable parameters, N and S, the latter a variable in which the sum is accumulated and ultimately returned to the main program. Since we need a loop, the variable I is declared as an INTEGER variable to serve as the loop index for the procedure. The main program also has a loop. For each INTEGER I between 1 and K it calls on PROCEDURE SUM to compute the sum from 1 to I and prints out the result. Again, a variable I is declared so it can be used as the index in the main program. *There are two different variables named I in this program.* One is used by the main program and one by the procedure. The one used by the main program is *global*; the other is *local* to the procedure. Alternatively, we say that the *scope* of the local variable is the procedure; that is the same thing as saying that the variable is only known within the procedure. The scope of a global variable is the entire program. It can be used by the main program or by any procedure (unless, of course, the same name is used within a procedure for a new variable—as was the case with the I in Example 11-1).

Let us look more closely at that example to see what else we can learn from it. Notice that there are actually three global variables—K, I, and TOTAL—and three variables local to the procedure—N, S, and I. I is the only name that is duplicated. This means that the other two global variables (K and TOTAL) can be referenced by the procedure as well as the main program.

In the example, we have provided the procedure access to the variable TOTAL by using it as an actual variable parameter. We weren't required to do this, since the procedure already had access to TOTAL. Here is Example 11-1 rewritten to take advantage of the fact that TOTAL is a global variable.

EXAMPLE 11-2

```

PROGRAM TRIV2 ( INPUT, OUTPUT );
VAR K, I, TOTAL: INTEGER;
  PROCEDURE SUM (N: INTEGER);
  VAR I: INTEGER;
  BEGIN
    TOTAL := 0;
    FOR I := 1 TO N DO
      TOTAL := TOTAL + I
    END;

  BEGIN (* TRIV2 *)
    READ (K);
    FOR I := 1 TO K DO
      BEGIN
        SUM (I);
        WRITELN (I, TOTAL)
      END
    END.
  
```

Although TRIV2 will work as well as TRIV1, it has the disadvantage that we cannot tell simply by looking at the procedure *heading* what variables in the main program will be altered by the procedure. This can make program bugs hard to locate. The use of global variables within a procedure also tends to destroy the generality of the procedure. (Note that, since the sum is always returned via the global variable TOTAL, we can no longer call this procedure from various points in a main program and request that the sum be returned via different variables.) In short, it is usually considered poor programming practice to use global variables within procedures.

Let us end this section with a slightly more complicated example in which, in addition to the main program, there are two procedures, one of which calls the other.

EXAMPLE 11-3

```

PROGRAM BILL ( INPUT, OUTPUT );
VAR I, CUSTNO: INTEGER; TOTAL, AMOUNT: REAL;
(* PROGRAM TO READ PURCHASES AND PRINT BILL *)
PROCEDURE STARS (N,M: INTEGER);
  
```



```

(* PRINTS N LINES OF M STARS *)
VAR I, J: INTEGER;
BEGIN
  FOR I := 1 TO N DO
    BEGIN
      FOR J := 1 TO M DO
        WRITE('* ');
      Writeln
    END
  END;
  (* *)
  PROCEDURE HEAD(CUST: INTEGER);
  (* PRINTS BILL HEADING, USING STARS *)
  BEGIN
    STARS(2, 50);
    Writeln;
    Writeln ('AMOUNT OWED BY CUSTOMER ': 33,
      CUST: 5);
    Writeln;
    STARS(2, 50)
  END;
  (* *)
  BEGIN (* MAIN PROGRAM *)
    READ(CUSTNO);
    TOTAL := 0;
    (* READ AND TOTAL 10 PURCHASES *)
    (* PRINT BILL. *)
    FOR I := 1 TO 10 DO
      BEGIN
        READ(AMOUNT);
        TOTAL := TOTAL + AMOUNT
      END;
    HEAD(CUSTNO);
    Writeln (TOTAL: 25: 2);
    Writeln;
    STARS (1, 10)
  END.

```

In this program, PROCEDURE STARS references the local names I, J, N, and M. PROCEDURE HEAD references one local name, CUST, as well as the procedure name STARS. STARS, which is also referenced by the main program, is a globally known name. (But just as STARS must precede the main program from which it is called, STARS must also precede HEAD in order for HEAD to call STARS. That is, the name STARS is not "known" until after it has appeared in a procedure heading.) The main program declares four variables, I, CUSTNO, TOTAL, and AMOUNT. None of these, although global, are referenced by the procedures. HEAD could, of course, reference any of them, while STARS could reference any except I.

Now suppose we do not want to call STARS from the main program. Suppose, in fact, we want STARS to be a facility known only to our header-printing PROCEDURE HEAD; that is, we want STARS to be a name local to HEAD. It turns out that we can arrange this. To explain how it is possible we will look more closely into the definition of a block. We will find that procedures can be *nested*, similarly to loops and IF statements. And we will find that this nesting has important implications for the notion of a “local” name. These matters are the subject of the next section.

11-6. BLOCK-STRUCTURED PROGRAMS

You will recall that the simple programs of the previous chapters consisted of a *heading* followed by a *block*. Similarly you have seen in this chapter that a procedure also consists of a heading followed by a block. Furthermore (except for Example 11-3) the programs you have seen in this chapter are all of the form

```
PROGRAM . . .  
VAR . . .  
PROCEDURE . . .  
    BEGIN  
        .  
        .  
        .  
    END;  
BEGIN  
    .  
    .  
    .  
END.
```

Is this still of the form of a heading followed by a block? Yes, only now the block is more complicated than a list of statements. In fact, it contains another block (the procedure block) *nested* within it. We can picture this structure as shown in Figure 11-1.

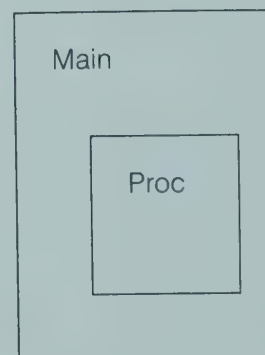


Figure 11-1

In Example 11-3, we nested two procedure blocks inside the main program, so that the structure is as shown in Figure 11-2.

We can also nest procedures inside procedures, since a procedure block, like the program block, can contain blocks within it. Thus, if we only want to call `PROCEDURE STARS` from `PROCEDURE HEAD`, we can nest `STARS` within `HEAD`, as shown in Figure 11-3.

So that you can see how this nesting is specified in a Pascal program, here is Example 11-3, rewritten so that the nesting is as shown in Figure 11-3. Now that `STARS` is inside `HEAD`, it cannot be referenced by the main program; it is, so to speak, hidden from the view of the main program.

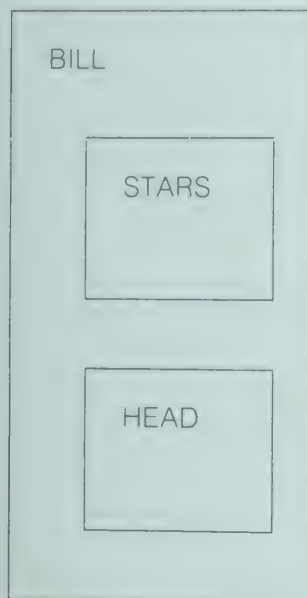


Figure 11-2

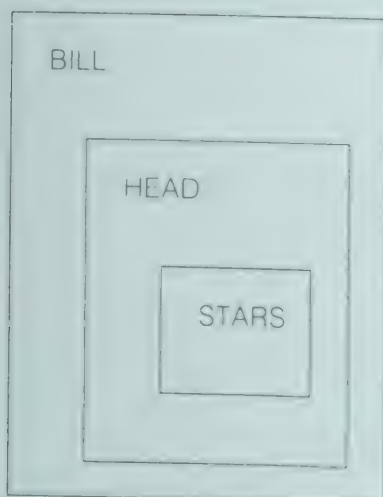


Figure 11-3


```

PROGRAM BILL (INPUT,OUTPUT);
VAR I,CUSTNO: INTEGER; TOTAL,AMOUNT: REAL;
PROCEDURE HEAD (CUST: INTEGER);
PROCEDURE STARS (N,M: INTEGER);
VAR I,J: INTEGER;
BEGIN
  FOR I := 1 TO N DO
    BEGIN
      FOR J := 1 TO M DO
        WRITE ('*');
      WRITELN
    END
  END;
BEGIN
  STARS(2,50);
  WRITELN;
  WRITELN('AMT OWED BY CUSTOMER ':33, CUST:5);
  WRITELN;
  STARS(2,50)
END;
BEGIN
  TOTAL := 0;
  READ(CUSTNO);
  FOR I := 1 TO 10 DO
    BEGIN
      READ(AMOUNT);
      TOTAL := TOTAL + AMOUNT
    END;
  HEAD(CUSTNO);
  WRITELN (TOTAL:25:2);
  WRITELN;
  WRITELN ('*****')
END.

```

Notice that to nest STARS within HEAD, we simply insert PROCEDURE STARS after the heading for HEAD and before the BEGIN that indicates the beginning of the executable statements in HEAD. The nested procedure would also follow the constant and variable declarations for HEAD, if there were any.

As the structure of programs becomes more complicated, the notions of “global” and “local” become more complicated too. A global name is still one that is known to the entire program, even to the most deeply nested procedure. Thus, in Example 11-4 the main program variables CUSTNO, TOTAL, and AMOUNT are all global variables. PROCEDURE HEAD has one local variable, CUST. Since STARS is nested inside HEAD, STARS can also ref-

erence CUST. (In Example 11-3, STARS could *not* reference CUST.) A name that is local to a procedure is also known to all of the procedures nested inside that procedure. But a name declared in a nested procedure is, of course, local to *that* procedure and not known to the procedure within which it is nested. Thus the variables declared within STARS—N, M, I, J—are local to STARS and cannot be referenced from outside STARS.

The block structuring of a program also has a critical effect on which other procedures a procedure may call on. In Figure 11-2, the names STARS and HEAD are known to the main program; hence they are global. STARS could call HEAD, just as HEAD calls STARS. This apparent symmetry is, however, complicated by the fact that a procedure cannot recognize the name of a procedure that has not yet been declared. We shall see in the next chapter how to get around this problem. In Figure 11-3 however, STARS is a name known only locally within HEAD. Thus, as we remarked above, the main program BILL can no longer invoke STARS.

In the general case there may be blocks nested within blocks nested within blocks—and so forth. You must keep in mind that a variable is not simply “local,” but *local to the particular procedure in which it is declared*. If all identifiers are different, then the *scope* of a variable X (or, more generally, any identifier) is the entire block for which X is defined, including any blocks

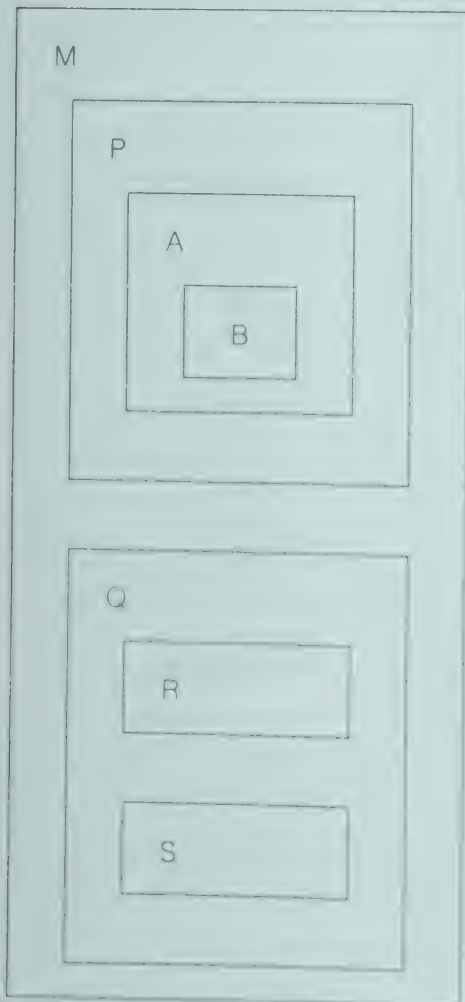


Figure 11-4

defined within that block. In Figure 11-4 we show a diagram of a fairly complicated block-structured program, showing the nesting of blocks within blocks. The blocks are labeled for reference as M, P, A, B, Q, R, and S. The objects (variables, arrays, etc.) declared for M (the main program block) are, of course, *global*. Their scope extends throughout M, P, A, B, Q, R, and S. The situation for objects declared for blocks nested within M is given in the following table.

Object declared for block	Scope
P	P,A,B
Q	Q,R,S
A	A,B
B	B
R	R
S	S

Keep in mind that this rule holds *when the names are different*. If a name used by a block duplicates a name used by a block which contains it, the inner block can no longer “see” the object declared in the outer block. (The inner block has preempted the object’s name, and no longer has a way to refer to that object.) Looking back at Figure 11-4, suppose Q and R both declare variables named V. Then Q’s V has scope Q, S; while the scope of R’s V is simply R.

You may be wondering at this point why blocks should be nested. The answer lies in the fact that, as we pointed out before, when one procedure is nested within another, it cannot be called—and hence utilized—by procedures outside the procedure or block containing it. Look again at Figure 11-4. The main program M utilizes (and hence calls) two procedures, P and Q. In turn, Q requires two procedures, R and S, to carry out its task. P, on the other hand, requires only A—but A calls upon B for aid. It is illegal for P to call B directly—or, for that matter, to call R or S, which are hidden inside Q. These rules encourage the programmer to arrange a set of procedures that call each other into a clear, logical program structure. The programmer should *not* scribble down code for a multitude of independent procedures and have them call each other in a random and confusing manner.

Formally, the rules for which procedures can call which other procedures are just part of the rules for the scope of identifiers. The name of a procedure is an identifier that has a scope like any other. The procedure heading is assumed to declare the procedure name for the block that contains that procedure. Thus the name of P is known (as it must be) to M—and also to Q. A and B, being in the same block as P, also know about P. But R and S do not.

This raises an interesting question. Since P necessarily knows its own name, can P call itself? The answer to this question is *yes*. Furthermore, A and B are also able to call P, and B can call A as well as vice versa. Procedures calling themselves (or calling other procedures which can, in turn, call them) present difficulties of implementation that caused early programming lan-

guages to simply forbid such behavior. And understanding what happens when a procedure calls itself is sufficiently difficult that we will postpone further discussion of this topic until the next chapter.

11-7 PROCEDURES WITH A SINGLE RESULT: FUNCTIONS

In many cases one needs to define procedures to compute a single result. For example, you might want to compute the square root of X , or perhaps X raised to the N th power. Or you might want a procedure to compute the area of a circle of radius R , or to determine the number of days between two given dates. This type of procedure is so common that it is given a special name. It is called a *function*, since it corresponds closely to the mathematical concept of function. There are especially easy ways provided in most computer languages to define and to invoke functions.

In Pascal, a function, like a procedure, consists of a *heading* and a *block*. The heading is only slightly different from that of a procedure. Here's the way we might head a function to calculate the area of a circle:

```
FUNCTION AREA (RADIUS : REAL) : REAL;
```

First, the heading identifies the block to follow as a FUNCTION named AREA. There then follows the parameter list. Finally, after the second colon, the *type* of the result is given. The result is essentially passed back to the calling procedure *via the name* AREA, as you shall see. The parameters in the parameter list should be *value* parameters (as in this example), since there is no need for functions to return results via variable parameters as procedures must do. (And a function should *not* change the values of its actual parameters!) Let us complete the function definition that we have begun.

```
FUNCTION AREA (RADIUS : REAL) :REAL;
CONST PI = 3.14159;
BEGIN
    AREA := PI * RADIUS * RADIUS
END;
```

Notice that, in a function, a memory location with the function name is set up, and the result of the computation must be assigned to that memory location by an ordinary assignment statement.

To use this function in a main program or procedure, all you have to do is to write, for example,

```
A := AREA(R)
```

The function AREA is invoked and the value of the actual parameter R is copied into the location named RADIUS for use by the function. The value of

AREA is then computed and handed back to the main program. Notice that the main program must then either assign the result to a location (in this case A) or use the result as part of a further computation, such as:

```
PIZZAPRICE := AREA(R) * PRICEPERSQIN
```

Here's a complete program giving another example of how we might use AREA(R) in calculating a further result.

```
PROGRAM CYLVOLUME (INPUT,OUTPUT);
  (* PROGRAM TO COMPUTE AND PRINT THE *)
  (* VOLUME OF A RIGHT CIRCULAR      *)
  (* CYLINDER, GIVEN THE HEIGHT H AND *)
  (* THE RADIUS R OF THE BASE.        *)
  VAR H,R,VOLUME:REAL;
  FUNCTION AREA (RADIUS:REAL) : REAL;
  CONST PI = 3.14159;
  BEGIN
    AREA := PI * RADIUS * RADIUS
  END;
  BEGIN (* MAIN PROGRAM *)
    READ (H,R);
    VOLUME := H * AREA(R);
    WRITELN ('VOLUME OF A CYLINDER WITH HEIGHT',
      H:15, ' AND BASE RADIUS', R:15, ' IS',
      VOLUME:15)
  END.
```

It may appear that in this context using a function is equivalent to the proverbial use of a sledgehammer to kill a gnat. You should imagine a more elaborate program, one which computes volumes and areas of geometrical figures of different shapes or perhaps uses geometrical calculations in solving an engineering problem. By providing ourselves with the function AREA—which can be used in any desired context—we have made the rest of our programming task that much simpler.

We should point out that, although a function computes and returns exactly one result, it may have more than one parameter. For example, here's a function that is handed three numbers, A, B, and C. It then determines the largest of the three and hands it back to the calling program.

```
FUNCTION MAX (A,B,C : REAL) : REAL;
  VAR T : REAL;
  BEGIN
    T := A;
    IF B > T THEN T := B;
    IF C > T THEN T := C;
    MAX := T
  END;
```


This function can then be inserted into the following program that reads three numbers and prints the maximum.

```
PROGRAM FINDMAX ( INPUT, OUTPUT );  
VAR X, Y, Z : REAL;  
(* INSERT FUNCTION MAX HERE *)  
BEGIN  
    READ (X,Y,Z);  
    WRITELN (MAX(X,Y,Z))  
END.
```

Since certain functions are used over and over by anyone who programs a computer, most programming languages provide some *standard* functions for you. These functions have been predefined, so that you can simply invoke them whenever you wish. You have already used two such functions—TRUNC and ROUND—supplied in Pascal. In Section 11-11 we will provide you with a lengthier list of standard Pascal functions. Also, in Section 11-10 we shall discuss some standard procedures that are available for your use. But first we will look at some other details of procedure and function usage.

11-8 PROCEDURES AND FUNCTIONS AS PARAMETERS

We have seen that a procedure or a function may have one or more parameters, which can be value parameters or variable parameters. Indeed, as illustrated in this chapter and the next, the parameter list provides a lot of flexibility in the usage of subprograms. In Pascal, further flexibility is provided by allowing procedure names and function names as parameters. The situation can be quickly illustrated by an analogy: Consider the family physician's sending her patient to a clinic as equivalent to a call to a procedure named CLINIC. Since she wishes the patient to be examined by a particular doctor in the clinic, one of the parameters for the call to PROCEDURE CLINIC is the doctor's name. The doctors in the clinic correspond to procedures that can be called by PROCEDURE CLINIC. Thus specifying a doctor's name is analogous to specifying the name of a procedure that is to be invoked to carry out its specialized task in PROCEDURE CLINIC.

In Pascal, there are, altogether, four kinds of parameters that can be specified in a parameter list: value parameters, variable parameters (prefaced by VAR), procedure parameters (prefaced by PROCEDURE), and function parameters (prefaced by FUNCTION and followed by a specification of the type of value returned). For example, a procedure might be headed by

```
PROCEDURE EX (FUNCTION F : REAL;  
              VAR X : INTEGER);
```

Thus procedure EX will make use of a function F. However, F is only a formal

parameter. The name of the actual function to be used will be supplied to EX by the calling program as the actual parameter corresponding to the formal parameter F. The only restriction on functions and procedures passed as actual parameters to other functions and procedures is that they can have *value parameters only*. (This rule makes the compiler's job a bit easier.)

The facility to specify functions or procedures as parameters is very useful in scientific or mathematical computations. For example, we might want to find a zero (or root) of a given function $f(x)$, that is, a value of x such that $f(x) = 0$. Now we could simply write a procedure that calls f , examines its values, and finds its zero. But it makes more sense to write a *general* procedure to find a zero of a function and then specify any *particular* function we are interested in as a parameter.

Let us illustrate what we mean by programming a particular method, known as the *bisection method*, for finding a zero of a continuous function. In this method, we start with the information that within some specified interval, say between a and b , the function has exactly one zero. Then we know that the function, let's call it f , looks something like the graph shown in Figure 11-5,

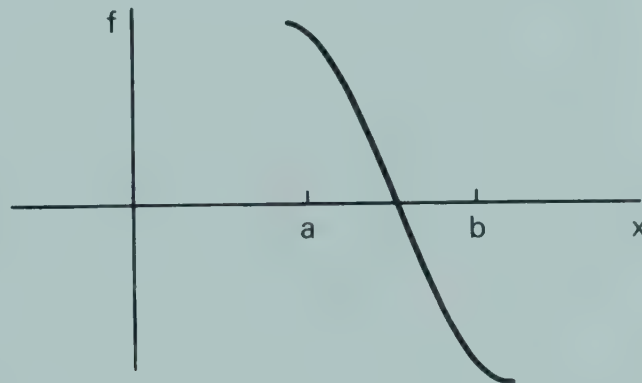


Figure 11-5

or maybe that shown in Figure 11-6.

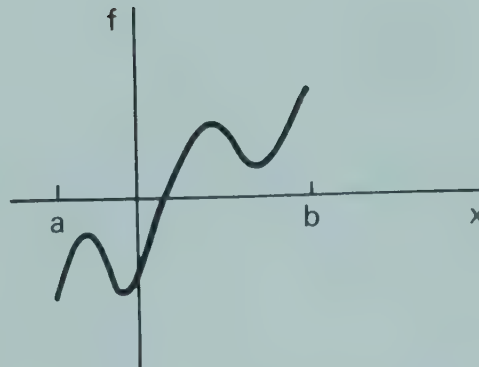


Figure 11-6

The important thing is that to the left of the zero f has one sign and to the right of the zero it has the opposite sign. To hunt for the value of x for which $f(x) = 0$, we proceed as follows. First, we divide the interval in half (*bisect* the

interval) and evaluate f at the midpoint m . If $f(m)$ has the same sign as $f(a)$, its sign must be opposite to that of $f(b)$, and we look for a zero in $[m, b]$. Again, we bisect the interval $[m, b]$, and look to see which half must have a zero. Continuing in this way we eventually home in on a sufficiently tiny subinterval containing a zero so that we can use any point of this subinterval as a good approximate value for the zero.

Let us see how we apply this method to finding a zero of $f(x) = x^4 - 2$ in the interval $[0, 2]$. Notice that $f(0)$ is negative, while $f(2)$ is positive. Evaluating $f(x)$ at the midpoint of $[0, 2]$ we find that $f(1) = -1$; therefore the zero must be in $[1, 2]$. Then we compute $f(1.5) = 3.0625$, so that the zero is in $[1, 1.5]$. And so forth. Here's a Pascal program to carry out this computation for us.

EXAMPLE 11-5

```

PROGRAM SPECIAL (INPUT, OUTPUT);
VAR A, B, ZERO, MIDPT, VALUE, FA : REAL;
    SIGN: BOOLEAN;
BEGIN
    A := 0.0; B := 2.0;
    FA := A*A*A*A - 2.0;
    SIGN := FA < 0.0;
    REPEAT
        MIDPT := (A+B)/2.0;
        VALUE := MIDPT*MIDPT*MIDPT*MIDPT-2.0;
        IF (VALUE < 0.0) = SIGN
            THEN A := MIDPT
            ELSE B := MIDPT
        UNTIL ABS(A-B) < 1.0E-010;
    WRITELN (MIDPT)
END.

```

(In examining this program, notice the use of the boolean variable SIGN to ensure that the function changes sign on successively chosen subintervals and the use of the standard function ABS to avoid worry over which interval endpoint is the larger.)

If you look at Example 11-5 carefully, you will see that very little needs to be done to turn it into a generally useful procedure for finding zeros of functions. The main alteration is to make the interval endpoints A , B and the function all parameters to be supplied to the procedure. Here's the result:

EXAMPLE 11-6

```

PROCEDURE ROOT (FUNCTION F: REAL;
                A, B: REAL;
                VAR ZERO: REAL);

```

```

VAR MIDPT, VALUE: REAL;
    SIGN: BOOLEAN;
BEGIN
    SIGN := F(A) < 0.0;
    REPEAT
        MIDPT := (A+B)/2.0;
        VALUE := F(MIDPT);
        IF (VALUE < 0.0) = SIGN
            THEN A := MIDPT
            ELSE B := MIDPT
    UNTIL ABS(A-B) < 1.0E-010;
    ZERO := MIDPT
END; (* ROOT *)

```

Now to find a zero of the function $x^4 - 2$ between $x = 0$ and $x = 2$, we insert PROCEDURE ROOT into the following program as indicated.

```

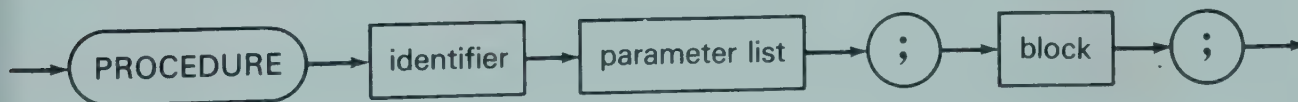
PROGRAM EX (INPUT, OUTPUT);
VAR Z : REAL;
FUNCTION FN(X : REAL) : REAL;
    BEGIN
        FN := X*X*X*X - 2.0
    END;
    (* INSERT PROCEDURE ROOT HERE *)
BEGIN (* MAIN PROGRAM *)
    ROOT (FN, 0.0, 2.0, Z);
    WRITELN (Z)
END.

```

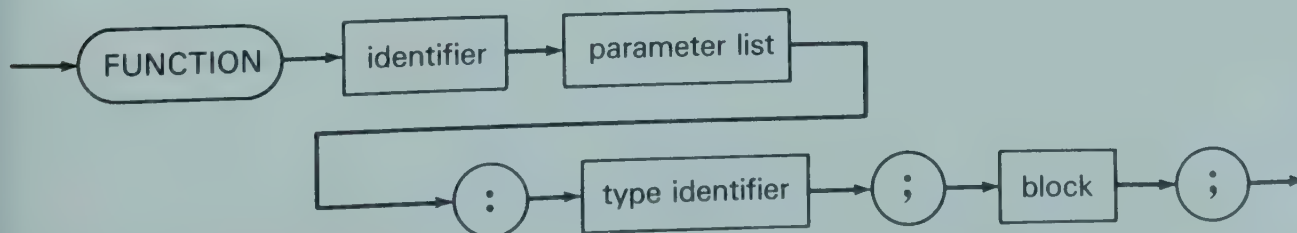
11-9 SYNTAX DIAGRAMS

First, let us look at the syntax for defining procedures and functions.

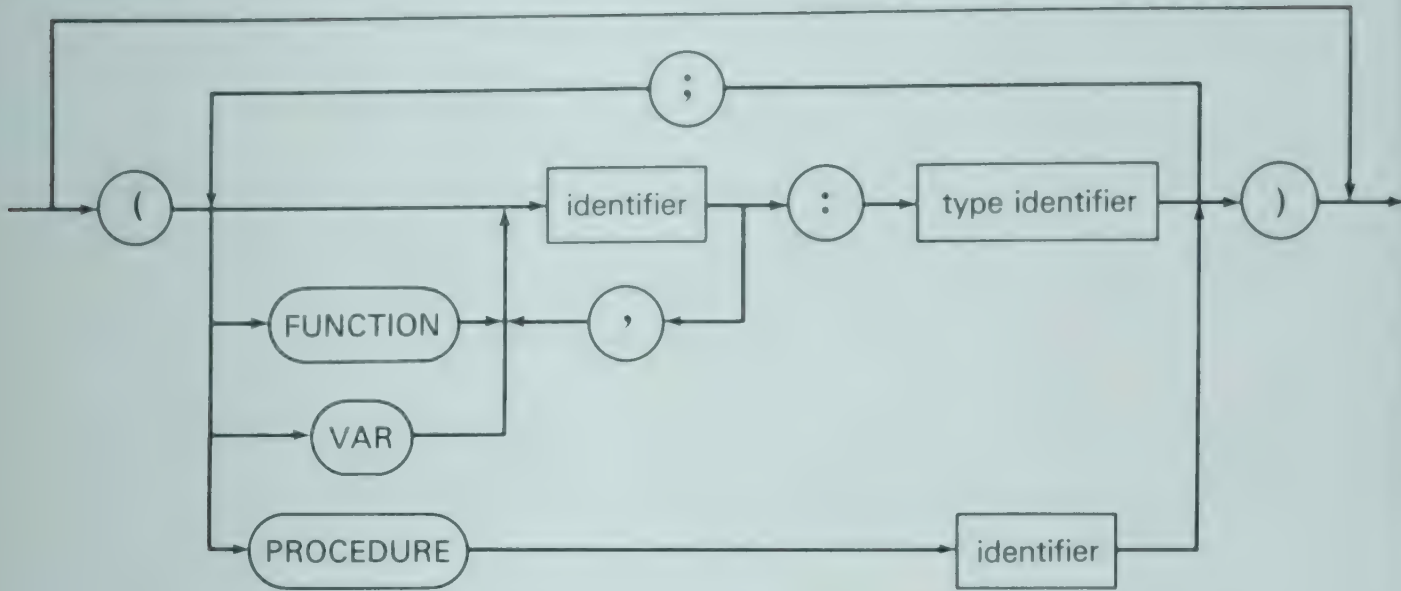
Procedure definition



Function definition



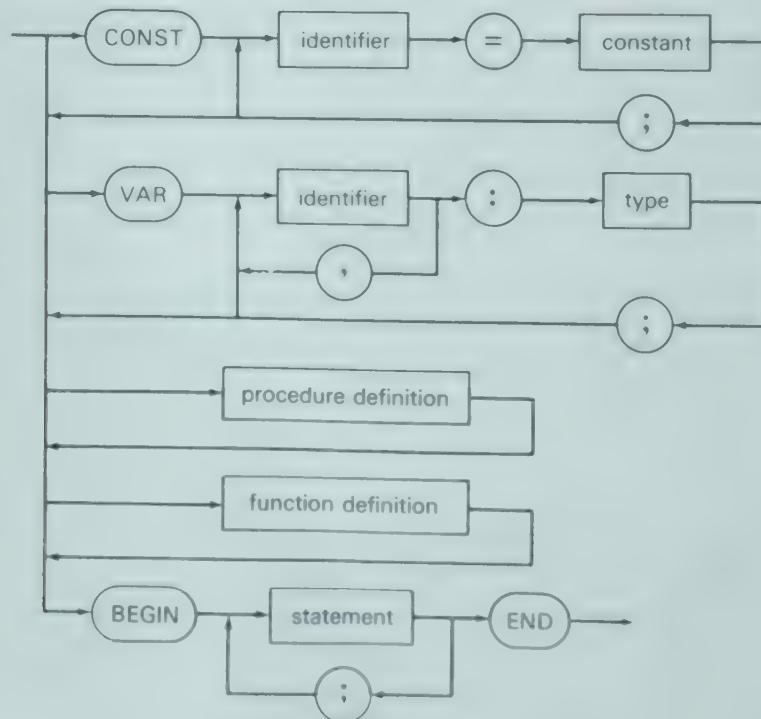
A *parameter list*, as used in this chapter, has the syntax:



Notice that the list may be empty; i.e., a procedure or function need not have any parameters. We will discuss later the formal difference between “type” and “type identifier.” For our present purposes, it suffices to note that a “type identifier” follows the same rules as any identifier, while a “type” can be more complicated (e.g., an array description).

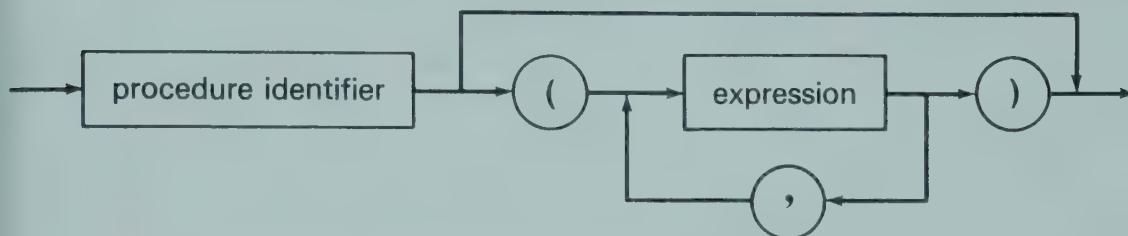
To see how procedure and function definitions fit into the general structure of a program, we will provide you with a more complete syntax diagram for *block* than we were able to present previously.

Block



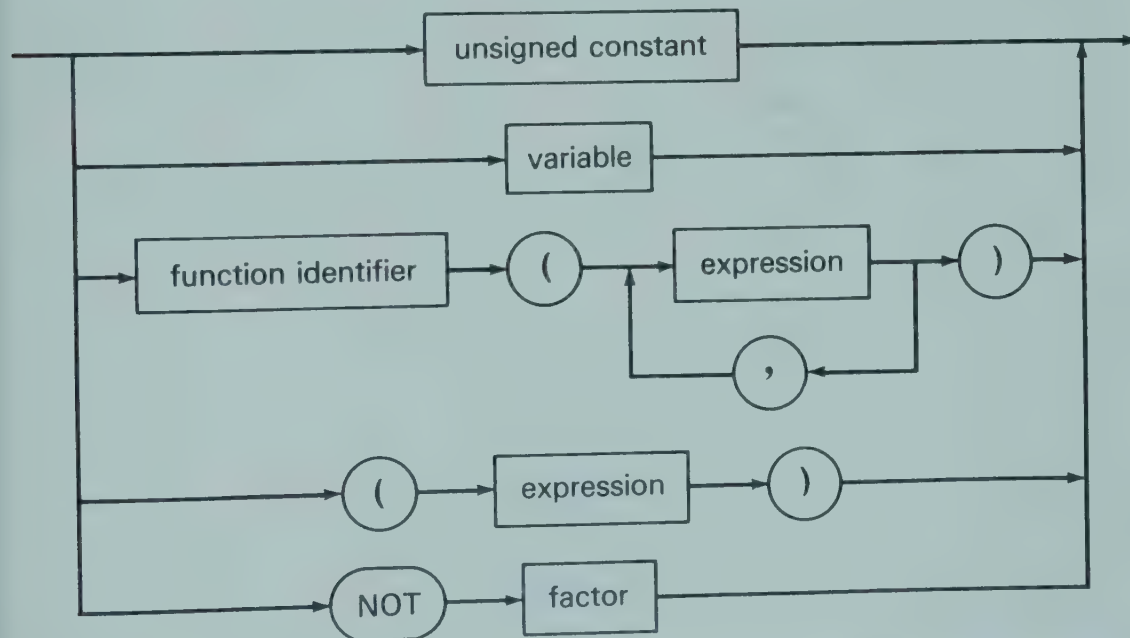
Notice that again we are dealing with *recursive* syntax—a block is defined in terms of procedure and function definitions, which in turn are defined in terms of blocks. It is easy to see that this is the formal syntax behind the block structure (blocks nested within blocks) of Pascal programs.

Procedures are invoked by means of *procedure statements* with syntax:



As usual, one must worry about meaning (semantics) as well as formal syntax. For example, you have seen that the list of expressions (actual parameters) in the procedure statement must agree with the formal parameters in the procedure parameter list with respect to number and type.

Finally, you have surely noticed that the invoking of a function fits rather differently into the Pascal language syntax than does a procedure call. In fact, a function call is a kind of factor. We here present a syntax diagram for *factor* which includes the function call.



This is still not the complete syntax for factor; one more possibility remains to be discussed later.

11-10 STANDARD PROCEDURES: INPUT-OUTPUT AND PACKING ARRAYS

As we mentioned earlier, the Pascal language has built into it a number of useful functions and procedures. Such procedures and functions are prede-

defined and are referred to as being “built-in” or “standard.” What happens is that when you use the name of a built-in function or procedure in your program—and you have not defined that name for some other purpose—the compiler recognizes that you want to use the standard procedure and automatically incorporates the code for that procedure in your program.

There are several standard procedures that you are already familiar with. These are the *procedures* READ, READLN, WRITE, and WRITELN. We did not want to confuse you earlier by telling you that the READ and WRITE “statements” introduced in Chapter 3 are actually procedure calls invoking the standard procedures READ, READLN, WRITE, and WRITELN.

Another useful procedure is PAGE. PAGE has no parameters: invoking it causes the printer to skip to the top of the next page. For example, if you need to print out a set of class rosters, you can call PAGE to begin each roster on a new page, so the lists can be separated and read more easily.

Two important standard procedures are PACK and UNPACK, which convert ordinary arrays into packed arrays, and vice versa. As we mentioned when we discussed character strings in Chapter 10, manipulating individual characters in a packed array is expensive. If you want to carry out a lot of such manipulations, what you should do is to call on UNPACK to unpack the array, operate on the resulting ordinary array, and then call on PACK to pack the altered array again.

The procedure UNPACK takes the characters in a packed array and places them in an ordinary array. It has three parameters:

1. The name of the packed array.
2. The name of the ordinary array.
3. An offset I—the first character in the packed array is placed in the Ith position in the ordinary array, the second character in the (I+1)st position, and so forth.

To be specific, suppose that two arrays have been declared:

```
VAR A: ARRAY [M..N] OF CHAR;
    PACKEDA : PACKED ARRAY [U..V] OF CHAR;
```

The statement

```
UNPACK (PACKEDA, A, I)
```

is equivalent to

```
FOR J := U TO V DO
    A[J-U+I] := PACKEDA[J]
```

PACK is also a three-parameter procedure. It takes characters from an ordinary array (beginning with the entry specified by offset I) and packs them

into a packed array. The parameters are listed in a different order from how they are in UNPACK, namely, (1) name of ordinary array, (2) offset, and (3) name of packed array.

Thus,

```
PACK(A, I, PACKEDA)
```

is equivalent to

```
FOR J := U TO V DO
    PACKEDA[J] := A[J-U+1]
```

11-11 STANDARD FUNCTIONS: ARITHMETIC AND ORDINAL FUNCTIONS

We have already discussed the functions ROUND and TRUNC, and the part they play in integer arithmetic. In addition, there are eight so-called *arithmetic functions* standard in Pascal to compute trigonometric functions, logarithms, etc. We have listed these, together with the rules for their use, in Table 11-1. To provide you with an example of their use, we include here a program which finds the real roots of quadratic equations using the functions SQR and SQRT.

Table 11-1

STANDARD ARITHMETIC FUNCTIONS

In all cases the parameter X is a real or integer expression.

Function	Computes	Type of result
ABS(X)	Absolute value of X	Same as X
SQR(X)	Square of X	Same as X
SIN(X)	Sine of X, where X is in radians	REAL
COS(X)	Cosine of X, where X is in radians	REAL
EXP(X)	The natural logarithm base (e) raised to the power X	REAL
LN(X)	Natural logarithm of X, if $X > 0$. (If $X \leq 0$, an error occurs.)	REAL
SQRT(X)	Positive square root of X, if $X \geq 0$. (If $X < 0$, an error occurs.)	REAL
ARCTAN(X)	Principal value, in radians, of the arctangent of X	REAL

EXAMPLE 11-7

```

PROGRAM QUADRATIC (INPUT, OUTPUT);
VAR A, B, C, X1, X2, DISC, RTDISC : REAL;
BEGIN
  READ (A, B, C);
  WRITELN ('A = ', A:15);
  WRITELN ('B = ', B:15);
  WRITELN ('C = ', C:15);
  WRITELN;
  IF A = 0
    THEN IF B = 0
      THEN IF C <> 0
        THEN WRITELN (' NO SOLUTIONS ')
        ELSE WRITELN (' ALL X SATISFY EQUATION ')
      ELSE BEGIN
        X1 := (-C)/B;
        WRITELN
          ('EQUATION OF LINE, ONLY ROOT IS',
           X1:15)
        END
      ELSE BEGIN
        DISC := SQR(B) - 4.0*A*C;
        IF DISC < 0
          THEN WRITELN ('ROOTS ARE COMPLEX')
          ELSE BEGIN
            RTDISC := SQRT(DISC);
            X1 := (-B + RTDISC)/(2.0*A);
            X2 := (-B - RTDISC)/(2.0*A);
            WRITELN ('ROOTS ARE: ', X1:15,
                     X2:15)
          END
        END
      END
    END
  END.

```

Pascal also includes several standard functions known as *ordinal functions*. These are mainly used for handling characters, although (as you will see in Chapter 13) they can also be used in certain other circumstances. The general idea of the ordinal functions is this. Associated with each character there is a nonnegative integer (technically an *ordinal number*) that gives the position of that character in the *collating sequence* (or ordering) of the characters. (See Section 10-2.) The ordinal functions allow us to obtain the ordinal number of a character, to obtain the character corresponding to a given ordinal number, and to obtain the characters which immediately precede or follow a given character in the sequence.

The function which gives the ordinal number of a character is ORD. Thus, for example, in a particular computer system:

The value of ORD('A') is 1.

The value of ORD('1') is 28.

The value of ORD('Z') is 26.

The value of ORD('*') is 39.

(You should try some examples to see what you get on your system.)

The function CHR is the *inverse* of ORD. That is, given an ordinal number, CHR will yield the corresponding character. Thus, for the same example as above,

CHR (1) is 'A'.

CHR (28) is '1'.

CHR (26) is 'Z'.

CHR (39) is '*'.

In fact, for any character value X the value of CHR(ORD(X)) will be the same as the value of X. We can also do more complicated operations. For example,

CHR (1 + ORD (' + '))

will yield the character following +, and

CHR (5 + ORD (' + '))

will yield the fifth character following +. (This presumes, of course, that + is not that close to the end of the sequence.) An easier way to find the character following a given one is to use the *successor function* SUCC. That is,

CHR (1 + ORD (' + '))

Similarly, there is a *predecessor function* PRED. PRED('+') will return the character preceding + in the collating sequence. (Note: if the specified character has no successor, then SUCC will give an error. Similarly, PRED will give an error when the parameter is the first character in the sequence.)

To demonstrate the usefulness of the ordinal functions, we have rewritten Example 10-3. Since ORD(CH) is just an integer, we can use ORD(CH) as a subscript to store the count for the character CH into its proper place in a array. Notice that the whole lengthy CASE statement of Example 10-3 is replaced by the simple statement

I := ORD (CH)

If, on your computer, `ORD('A')` is not 1 but, say, 25, you would have to adjust the array declaration to read

```
... ARRAY [25..50]...
```

and then adjust the FOR loops accordingly.

```
PROGRAM CNTTEXT2 (INPUT,OUTPUT);
VAR CH: CHAR;
    I: INTEGER;
    COUNTS: ARRAY[1..26] OF INTEGER;
BEGIN
  FOR I := 1 TO 26 DO
    COUNTS[I] := 0;
  REPEAT
    READ (CH);
    IF (CH >= 'A') AND (CH <= 'Z')
    THEN
      BEGIN
        I := ORD(CH);
        COUNTS[I] := COUNTS[I] + 1
      END
    UNTIL CH = '*';
  FOR I := 1 TO 26 DO
    WRITELN (I,COUNTS[I])
  END.
```

11-12 A PROGRAMMING PITFALL: SIDE EFFECTS

You are all aware of the fact that actions you take in your daily lives can have undesirable side effects. Enjoying too many hot fudge sundaes can make you fat. Cheating on exams can get you thrown out of school. The side effects that a computer programmer must worry about are unintended changes in the values of nonlocal variables by a procedure. One way to cause inadvertent side effects is to use a variable parameter where you should have used a value parameter. Example 11-8 provides an illustration. Here the procedure `BAD`, when supplied with a value for `N`, computes the value of the expression $2 + (N + 1)^6$ and assigns it to the variable parameter `ANSWER`. As a first step the procedure increments `N`, so that its value is `N + 1`. The problem is that, since `N` is a variable parameter, `N` and `ONE` (the actual parameter) refer to the same memory location. Thus, when `N` is incremented so is `ONE`. The output of this program is

THE VALUE OF ONE IS 2

This is distressing news if we planned to use `ONE` elsewhere in our program'

EXAMPLE 11-8

```

PROGRAM ONETWO ( INPUT,OUTPUT );
VAR ONE,A: INTEGER;
PROCEDURE BAD (VAR N,ANSWER : INTEGER);
BEGIN
    N := N + 1;
    T := SQR(N);
    ANSWER := 2 + T * T * T
END;
BEGIN
    ONE := 1;
    BAD(ONE,A);
    WRITELN ( 'THE VALUE OF ONE IS',ONE:2)
END.

```

Another way to cause side effects is to neglect to declare local variables in a procedure. For example, here's a procedure that interchanges two integers. It will perhaps be useful in a sorting program.

```

PROCEDURE SWAP(VAR A,B : INTEGER);
BEGIN
    TEMP := A;
    A := B;
    B := TEMP
END;

```

The mistake we have made here is to forget to declare TEMP as a local variable. Since TEMP is a useful name for a temporary storage location, the main program may well have a variable named TEMP. In this case, the compiler will think that we *meant* to use the global TEMP in SWAP and will not give us an error message. Consequently, the value of the variable TEMP in the main program will be changed. The ultimate results of this side effect could be severe—and the bug could be hard to find.

In summary, to avoid side effects:

1. *Declare all formal parameters as value parameters, unless the parameter is to be used to return a result to the calling program.* (There is an exception to this rule. You will see later that parameters are not necessarily variables. For example, they can be arrays. And you would probably want to think twice before copying a large array by declaring it as a value parameter.)
2. *Don't forget to declare all local variables. And avoid the temptation to use global variables in a procedure just to save yourself a bit of writing.*

If the parameter list of a procedure contains several variable parameters V1,V2,...VN, then the corresponding *actual parameters should be distinct*.*

*On the other hand, make sure you understand that, for value parameters, there is no problem if the corresponding actual parameters are not distinct.

If they are not distinct, then the procedure will be using the same memory location for more than one purpose, and the result is likely to be bizarre. See if you can fight your way through the confusion to determine the output of this example:

```

PROGRAM SEESAW1 ( INPUT ,OUTPUT );
VAR I: INTEGER;
PROCEDURE FAKE1 (VAR SEE,SAW: INTEGER);
BEGIN
    SEE := -1;
    SAW := -SEE
END;
BEGIN
    I := -1;
    FAKE1(I,I);
    Writeln (I)
END.

```

If one of these parameters is made a value parameter (so that the procedure is working with a copy), then the procedure is not overwriting the same location in quite so muddled a fashion. But the program can be nearly as confusing, as you can see by working through the following:

```

PROGRAM SEESAW2 ( INPUT ,OUTPUT );
VAR I: INTEGER;
PROCEDURE FAKE2 (VAR SEE: INTEGER; SAW: INTEGER);
BEGIN
    SEE := -1;
    SAW := -SEE
END;
BEGIN
    I := -1;
    FAKE2(I,I);
    Writeln (I)
END.

```

(You should run these examples on the computer to check that you have correctly understood what happens. Insert extra Writeln statements, if you need to, to follow through the changes in the variables.)

11-13 PROGRAMMING

HINTS: TESTING AND DEBUGGING

A MODULAR PROGRAM

One advantage of writing a program as a set of procedures is that we can test and debug each procedure separately. To return to our medical analogy, the

family doctor relies on the fact that the x-ray technicians and the lab personnel have been thoroughly tested and properly certified *before* she calls upon them to carry out their tasks. Similarly, before a procedure is incorporated into a large, complicated program, it should be tested and debugged with a small *test driver program*. The driver program is a very simple main program that invokes the procedure for given sets of parameters and writes the results that are returned. In this way you can satisfy yourself that a procedure is working correctly *before* you put it into your program.

If each module of your program has been carefully tested and debugged, there's a reasonable chance that the whole program will work well when everything is put together. It still needs testing, however. But if you have heeded our warnings and not written procedures that cause mysterious side effects, the difficulties of this final stage of testing and debugging should be minimal.

11-14 EXERCISES

- 11-1.** What does the following program print? What are the formal and actual parameters of the procedure?

```
PROGRAM NUMBERS ( INPUT, OUTPUT );
  VAR I, OUT : INTEGER;
  PROCEDURE ABSOLUTE (K: INTEGER; VAR L: INTEGER);
  BEGIN
    IF K >= 0 THEN L:= K
      ELSE L:=-K
  END;
  BEGIN
    FOR I:= -2 TO 2 DO
      BEGIN
        ABSOLUTE(I, OUT);
        WRITELN(OUT)
      END
    END.
  END.
```

- 11-2.** What is printed by the following?

```
PROGRAM S( INPUT, OUTPUT );
  VAR X: INTEGER;
  PROCEDURE P(A: INTEGER);
  BEGIN
    WRITELN (A+X)
  END;
  BEGIN
    X:=10;
    P(X)
  END.
```

11-3. What is printed by the following?

```

PROGRAM C (INPUT, OUTPUT);
VAR M, N, R : INTEGER;
    PROCEDURE EX (VAR T, U, M : INTEGER);
    BEGIN
        T := U + M;
        U := U + 1
    END;
BEGIN
    M := 48;
    N := 18;
    R := 3;
    WRITELN (M:2);
    EX (M,N,R);
    WRITELN (M:2);
    EX (M,N,R);
    WRITELN (M:3, N:3, R:3)
END.

```

What is printed if T is a value parameter? If U is a value parameter? If both T and U are value parameters?

11-4. What is printed by the following?

```

PROGRAM SB (INPUT, OUTPUT);
VAR M, N : INTEGER;
    PROCEDURE REM (VAR M, N, R : INTEGER);
    BEGIN
        R := M - N;
        WRITELN (M:3, N:3, R:3)
    END;

    PROCEDURE F (VAR L, M : INTEGER);
    VAR R : INTEGER;
    BEGIN
        REM (L,M,R);
        WHILE R >= M DO
            BEGIN
                L := R;
                REM (L,M,R)
            END
        END;
    END;

BEGIN
    M := 48;
    N := 18;

```

```

      F (M,N) ;
      WRITELN (M,N)
END .

```

- 11-5.** Write a function that finds the area of a triangle with sides of lengths A, B, and C (REAL values). The formula to use is

$$\text{AREA} = (1/4) \sqrt{(A+B+C)(-A+B+C)(A-B+C)(A+B-C)}$$

- 11-6.** Write a function that tests an integer value to determine whether or not it is prime. The function should return TRUE if the integer is prime and FALSE if it is not. Use this function in an interactive program that prompts the user to type an integer and then tells him whether it is prime.

- 11-7.** Write a procedure to print all primes from 1 to N.

- 11-8.** Write a procedure to compute the length of the hypotenuse of a right triangle given the lengths A and B of the other two sides. (Recall that $\text{hypotenuse} = \sqrt{A^2 + B^2}$.) Use this procedure in a program to compute the diagonal of a box, given its three dimensions: HEIGHT, LENGTH, and WIDTH.

- 11-9.** Study the program QUADRATIC in Section 11-11. Draw a flowchart for it. Next, expand the flowchart to compute and print complex roots as well. Then incorporate the complex-root solver into the program. The output in case of complex roots should be something like

```

ROOTS ARE :
    495.22 + 25.0I
    495.22 - 25.0I

```

Finally, write a set of test data for this program and *thoroughly* test it out. (Note the many possible cases that need to be checked.)

- 11-10.** Write a procedure to find the next Friday the 13th, given today's date and the day of the week. (You can ignore the problem of leap years.)
- 11-11.** Write a program to compute and print the number of days between two given dates. Example: Given the dates 5 30 1979 and 6 25 1979, the program should print "The number of days from May 30, 1979, to June 25, 1979, is 26."
1. Don't forget about leap years. And so that you don't have to worry about the leap-year rules for century years, you may want to restrict your program input to dates between 1901 and 1999.
 2. Depending on the value of MAXINT for your computer (see Exercise 8-4), you may need to restrict the range of allowable dates still further.
 3. Notice that the names of the months are printed. You can handle this by the same scheme we used at the end of Section 9-5 to write the names of the days of the week. But turn it into a procedure so you don't have to type the CASE statement into your program twice.
 4. Another useful procedure would be one which, given the number identifying a month, returns the number of days in that month.

- 11-12.** Suppose we have a standard alphabet of n characters and a key alphabet which is a permutation of the standard alphabet. We can encode a message written in the standard alphabet as follows. If the i th character in the message is the j th character in the standard alphabet, we replace it by the $(i+j)$ th character in the key alphabet. Since $i+j$ can be greater than n , we treat the key alphabet as though it were periodic. Thus the i th, $(n+i)$ th, $(2n+i)$ th, $(3n+i)$ th, ... characters are identical. For example, suppose $n = 40$ and the standard and key alphabets are:

Standard	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 ..
Key	Q,IL3X.ZAT09KN'8BH7C1YRDM4PF2 GOVJE6US5W

If the 100th character in the message is the 25th character in the standard alphabet, Y, it is replaced by the 125th or 5th character in the key alphabet, 3. Write a program ENCODE that reads a 50-character message string and encodes it, using the standard and key alphabet strings STD and KEY, as given above.

SUMMARY OF CHAPTER 11

The definition of one or more procedures may be included in a program block right after the VAR declarations. A procedure definition begins with a heading that contains the word PROCEDURE, followed by the procedure name and then a list of formal parameters enclosed in parentheses. The body of the procedure is a block. This block, like the program block, may contain other procedures as well as declarations and executable statements. Thus procedures may be nested within procedures. To call a procedure, we simply write down the procedure name together with a list of actual parameters enclosed in parentheses.

A procedure may have two different kind of parameters, value parameters and variable parameters. The value of a value parameter cannot be altered by the procedure, since the procedure has access only to a copy of the value. The value of a variable parameter may be altered by the procedure, since the procedure has access to the actual parameter.

A global name is a name that is known to the main program and to all procedures that do not have the same name as a local name. A local name is a name that is known to a particular procedure (including procedures nested within it if they do not define that name locally) but is not known outside that procedure.

Conceptually, a function is merely a special case of a procedure. A function computes a single value, which is returned to the calling program via the name of the function. That is, to invoke a function you simply include the function name followed by the list of actual parameters (in parentheses) in any expression.

Function definitions are included in program (or procedure) blocks after any procedure definitions. The heading of a function follows the form:

FUNCTION F(parameter list): type;

where the type is the type of value computed as F.

Many standard functions and procedures are built into your Pascal compiler and available for your use. Examples are the procedures READ, READLN, WRITE, and WRITELN.

RECURSIVE SUBPROGRAMS

12-1 INTRODUCTION

One lesson we learned in the preceding chapter on subprograms is how to delegate responsibility (or how to “pass the buck”). When the family doctor is asked to examine a patient, she will seek the assistance of the radiologist, the heart specialist, and so on. When the Department of Energy is asked for a comprehensive plan for the conservation of energy, it will ask the Department of Transportation, the Department of Defense, and probably every other department in the government to submit their opinions on the situation. The following conversation is another example in the same spirit:

“How old is your son David, Mr. Jones?”

“Oh, he is two years younger than the Wongs’ boy, Robert, next door.”

Mr. Jones did give a satisfactory answer to the question, since if we can find out Robert’s age, we can immediately determine David’s age. But then the following conversation may subsequently take place:

“How old is your son Robert, Mr. Wong?”

“Oh, he is three years younger than the Hansons’ girl, Rebecca, next door.”

Assuming that Hanson will give us a straight answer, the example is analogous to the following sequence of subprogram calls. Imagine that Jones is a subprogram that is called upon to determine the age of David. The subprogram Jones calls another subprogram Wong to find out the age of Wong’s son Robert. The subprogram Wong calls on subprogram Hanson to tell the age of Rebecca. After Hanson returns Rebecca’s age to Wong, Wong can compute

Robert's age and hand it back to Jones. Jones is then able to compute David's age and return it to the calling program. Figure 12-1 illustrates the sequence.

But now consider the following conversation with Mr. Charlie Chan, our wise detective from the orient.

"How old is your number 3 son, Mr. Chan?"

"Oh, he is two years younger than number 2 son."

"But, how old is your number 2 son?"

"He is three years younger than number 1 son."

"But, how old is your number 1 son?"

"Let me see, he was born in the Year of the Dog. He must be 27 now."

Now we have information from which we can determine that number 2 son is 24 years old and number 3 son is 22 years old. In this example, when Mr. Chan was asked the age of his number 3 son, he, like Jones and Wong, tried to let somebody else answer the question for him, except this "somebody else" turned out to be himself. Diagrammatically, we have the situation shown in Figure 12-2.

Again, imagine that Mr. Chan is a subprogram. We then have a situation in which a subprogram calls on itself. Indeed, Pascal subprograms are allowed to call on themselves. A call of a subprogram to itself is known as a *recursive call*; a subprogram that calls itself is termed *recursive*.

Let us examine this example a little closer. A subprogram (Mr. Chan) is called upon to perform a certain task (to determine the age of his number 3 son). The subprogram calls on itself to perform a similar task (to determine the age of his number 2 son), knowing that the completion of this task will enable him to complete the original task. However, in order to determine the age of number 2 son, the subprogram needs to call on itself to find out the age of the number 1 son. Note that such a sequence of events will terminate only when, at a certain point, the subprogram is able to complete a task it is called upon to perform without further calling on itself. (Mr. Chan can determine

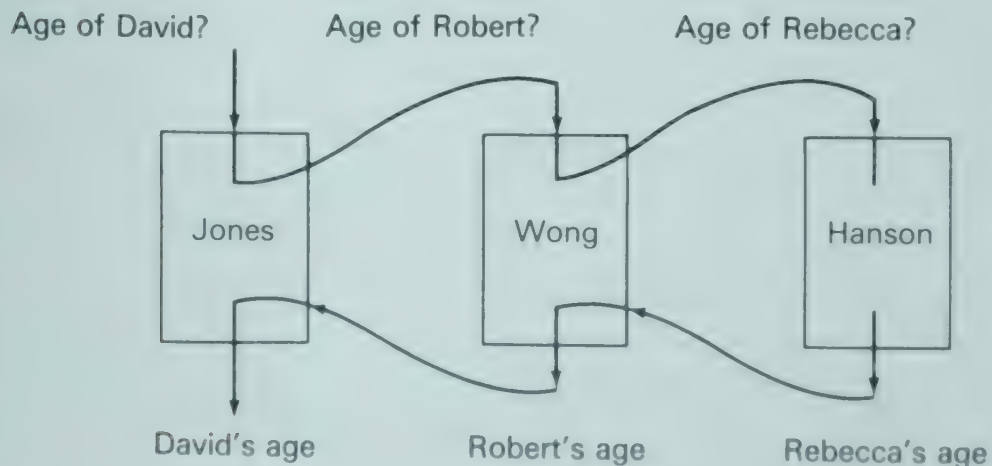


Figure 12-1

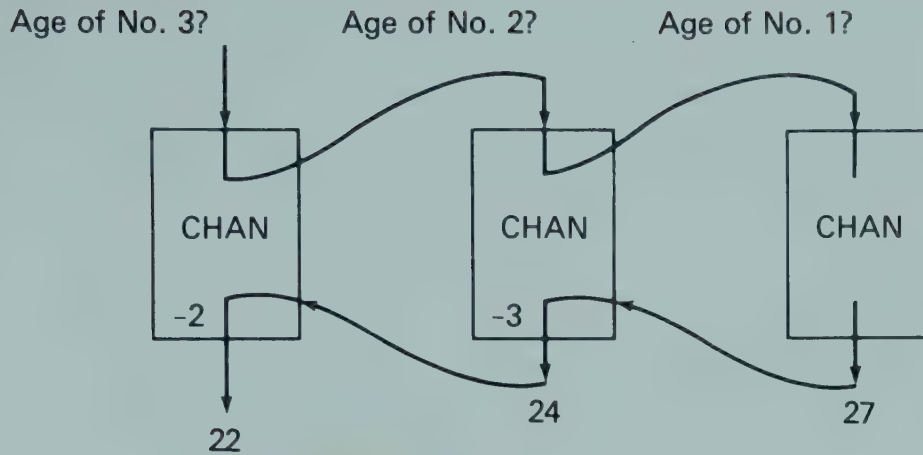


Figure 12-2

the age of his number 1 son without referring to the age of another son.) Indeed, any subprogram that calls on itself must contain an “escape route” (a *termination condition*), so that the subprogram will not repeatedly call on itself forever.

One can imagine that when a subprogram calls on itself again and again, some kind of careful bookkeeping is necessary so that no confusion will arise. Happily, all the details are handled automatically.

12-2 A SIMPLE EXAMPLE

Suppose the salaries of the employees of a company are increased according to the following formula: Each year, an employee receives a 5 percent raise plus a flat \$500 increase. Suppose an employee starts with a salary of \$10,000 per year; we want to know how much he makes in the tenth year. In general, let a_n denote the employee's salary in the n th year. Then we have the relationship

$$a_n = \begin{cases} 10,000 & (\text{for } n = 1) \\ 1.05 a_{n-1} + 500 & (\text{for } n \geq 2) \end{cases}$$

That is, in order to determine a_{10} , we must know the value of a_9 . In order to determine a_9 , we must know the value of a_8 , and so on. The sequence of computations can be depicted as shown in Figure 12-3. Each of the boxes in Figure 12-3 represents the execution of a program, albeit a very simple one, that computes the salary in the n th year when given the salary in the $(n-1)$ st year—except for the last box, which tells us the salary specified for the first year. The similarity between Figure 12-2 and Figure 12-3 will certainly not have

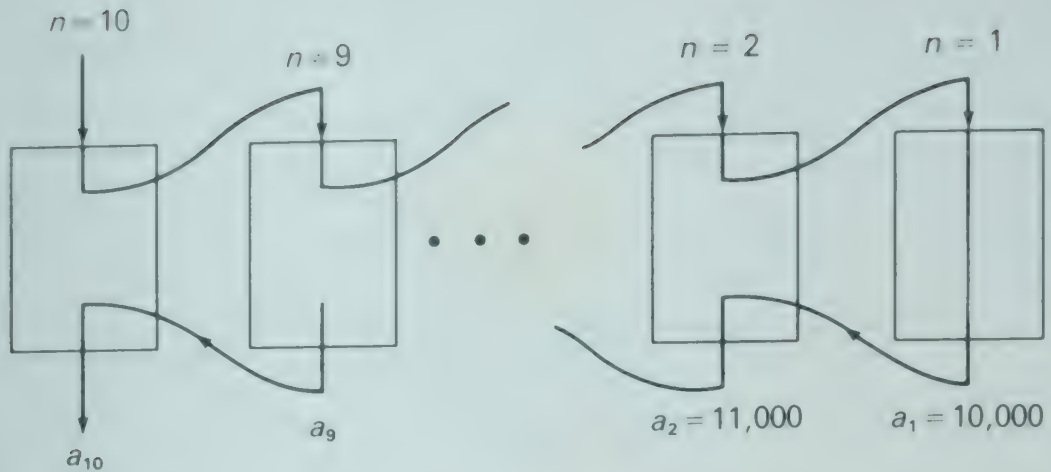


Figure 12-3

escaped the reader. Indeed, we immediately realize that we do not need 10 distinct programs to carry out the computation shown in Figure 12-3; a simple program that calls on itself will do. Here is a Pascal procedure that will do the job.

EXAMPLE 12-1

```

PROCEDURE COMPSAL (N: INTEGER; VAR SALARY: REAL);
BEGIN
  IF N = 1 THEN SALARY := 10000.0
  ELSE
    BEGIN
      COMPSAL (N-1, SALARY);
      SALARY := 1.05 * SALARY + 500.0
    END
END;

```

Notice how this procedure works. First it looks to see whether it is dealing with the special case (the termination condition) $n = 1$. If it is, it returns the value 10000 for SALARY. But if it is not, it proceeds to request the salary a_{n-1} for year $n - 1$ by the recursive call

COMPSAL (N-1, SALARY)

When this salary value is returned (in general after further recursive calls) the procedure uses the given formula to compute the salary a_n for the year n from the value a_{n-1} it has obtained by calling itself.

Note that in Example 12-1 we have made N a value parameter because we want only a copy of the current value of N to be handed to COMPSAL each time it calls itself. But SALARY must be a variable parameter, so that the successive values computed for it are handed back to each calling program in turn.

In the example below, we have included COMPSAL in a complete program that reads an integer K, then calls COMPSAL to compute the salary for the Kth year. In addition to printing the final answer in the main program, we have included a WRITELN statement in the procedure, so that the successive values computed for SALARY are clearly visible.

EXAMPLE 12-2

```
PROGRAM TCSAL ( INPUT , OUTPUT ) ;
VAR K : INTEGER ;
    S : REAL ;
PROCEDURE COMPSAL ( N : INTEGER ; VAR SALARY : REAL ) ;
BEGIN
    IF N = 1 THEN SALARY := 10000.0
    ELSE BEGIN
        COMPSAL ( N-1 , SALARY ) ;
        SALARY := 1.05 * SALARY + 500.0
    END ;
    WRITELN ( 'SALARY FOR N =', N:3, ' IS',
              SALARY:12:2 )
END ;

BEGIN
    READLN ( K ) ;
    COMPSAL ( K , S ) ;
    WRITELN ;
    WRITELN ( 'ANSWER FOR K =', K:3, ' IS', S:12:2 )
END .
```

When we ran this program with K = 10, the output was:

```
SALARY FOR N = 1 IS 10000.00
SALARY FOR N = 2 IS 11000.00
SALARY FOR N = 3 IS 12050.00
SALARY FOR N = 4 IS 13152.50
SALARY FOR N = 5 IS 14310.12
SALARY FOR N = 6 IS 15525.63
SALARY FOR N = 7 IS 16801.91
SALARY FOR N = 8 IS 18142.01
SALARY FOR N = 9 IS 19549.11
SALARY FOR N = 10 IS 21026.56

ANSWER FOR K = 10 IS 21026.56
```


It may have occurred to you that we didn't really need to use a recursive procedure to do this salary computation. The following piece of code gets the same job done.

```
SALARY := 10000.0;  
FOR N := 2 TO K DO  
    SALARY := 1.05 * SALARY + 500.0
```

Here, instead of successively inquiring about the preceding year's salary until we get back to year 1, we have just started with year 1 to begin with. The actual computation from then on—computing each year's salary from the preceding one—is identical for the two approaches. Why then do we want to write recursive procedures? The answer is that for some computations it is much more natural to use the recursive approach. Indeed, sometimes it is extremely difficult to know ahead of time exactly where the successive recursive calls will lead us and hence to define a starting point. And even if we have a starting point, converting the recursive approach into a FOR loop is often a lot more difficult than the simple example of this section would seem to indicate. In the next section we give an example of a computation for which the recursive approach is the natural one. After you study this example, think about how you might try to turn the recursive procedure around into a loop that does the same thing.

12-3 A PRACTICAL EXAMPLE

Suppose we are given an array of eight numbers $A[1]$, $A[2]$, ..., $A[8]$, and we are asked to find the largest and the smallest of these. A straightforward way to do this is to examine the numbers one by one as Example 12-3 illustrates. (Note that the elements of the array A are accessed as global variables.)

EXAMPLE 12-3

```
PROCEDURE MAXMIN (VAR LARGEST, SMALLEST : REAL);  
VAR I : INTEGER;  
BEGIN  
    SMALLEST := A[1];  
    LARGEST := A[1];  
    FOR I := 2 TO 8 DO  
        BEGIN  
            IF A[I] > LARGEST  
            THEN LARGEST := A[I]  
            ELSE IF A[I] < SMALLEST  
            THEN SMALLEST := A[I]  
        END  
    END;  
END;
```

There is an alternative way to solve the problem. Let us divide the eight numbers into two groups $A[1] \dots A[4]$ and $A[5] \dots A[8]$. Suppose we can find the largest and the smallest of $A[1] \dots A[4]$; let us call them **LARGEST1** and **SMALLEST1**. Also suppose we can find the largest and the smallest of $A[5] \dots A[8]$; let us call them **LARGEST2** and **SMALLEST2**. Clearly, the larger of **LARGEST1** and **LARGEST2** is the largest of $A[1] \dots A[8]$, and the smaller of **SMALLEST1** and **SMALLEST2** is the smallest of $A[1] \dots A[8]$. Of course, the question is now to determine **LARGEST1**, **SMALLEST1**, **LARGEST2**, and **SMALLEST2**, that is, to solve two problems of the same form as the original one, but for arrays that are only half as long. The perceptive reader will guess that we shall further divide $A[1] \dots A[4]$ into two groups and $A[5] \dots A[8]$ into two groups, and so forth. The process of dividing large groups into small groups will eventually terminate when each group has only one or two members. (A group of three, for example, will be divided into one group with one member and one with two.) The larger and smaller of each group of two is then immediately found by a single comparison, while the largest and smallest of a group of one are both the one member itself. The following outline indicates the major steps in implementing this method to determine the largest and the smallest among the numbers $A[\text{FIRST}] \dots A[\text{LAST}]$.

1. Compute **MIDDLE** as $(1/2) * (\text{LAST} - \text{FIRST}) + \text{FIRST}$.
2. Find the largest and the smallest of $A[\text{FIRST}] \dots A[\text{MIDDLE}]$; these will be denoted **LARGEST1** and **SMALLEST1**.
3. Find the largest and the smallest of $A[\text{MIDDLE} + 1] \dots A[\text{LAST}]$; these will be denoted **LARGEST2** and **SMALLEST2**.
4. Let **LARGEST** be the larger of **LARGEST1** and **LARGEST2**. Let **SMALLEST** be the smaller of **SMALLEST1** and **SMALLEST2**.

Notice that as long as a group has more than two members steps 2 and 3 will involve recursive calls to the procedure itself; i.e., to begin at step 1 and do the next breakdown. We now work out the details of the procedure. First, what should the parameters be? Clearly, the procedure needs to return **LARGEST** and **SMALLEST**; we shall define variable parameters with these names. Also, the procedure needs to be given the subscript range to be operated on; that is, if the procedure is to return the largest and smallest elements of $A[\text{FIRST}] \dots A[\text{LAST}]$, it requires the values of the integers **FIRST** and **LAST**. In short, the heading of the procedure will be

```
PROCEDURE MAXMINR ( FIRST , LAST : INTEGER ;
  VAR LARGEST , SMALLEST : REAL ) ;
```

(Again, we assume that we operate on a globally declared array **A**.) Now let us look more closely at what the procedure must do. There are three cases:

1. If **FIRST** = **LAST**, the array to be examined has only one element. So **SMALLEST** = **LARGEST** = $A[\text{FIRST}]$.
2. If **LAST** = **FIRST** + 1, the array has two elements. A simple comparison tells us which to return as **SMALLEST** and which as **LARGEST**.

3. If $LAST > FIRST + 1$, the array has at least three elements. In this case we divide the array in half (approximately), call MAXMINR to find the SMALLEST and LARGEST for each half, and combine the results by comparing the two values of SMALLEST and the two values of LARGEST.

In this outline we see the important features of recursive subprograms. Cases 1 and 2 represent the simple situations (termination conditions) for which explicit values are returned. Case 3 is the general, recursive step. The complete procedure is given below as Example 12-4. Notice that we have added an extra test at the beginning to make sure that the last subscript LAST of the subarray is not less than the first subscript FIRST.

EXAMPLE 12-4

```

PROCEDURE MAXMINR (FIRST, LAST: INTEGER;
                   VAR LARGEST, SMALLEST: REAL);
VAR MIDDLE: INTEGER;
    L1, (* LARGEST OF FIRST SUBARRAY *)
    S1, (* SMALLEST OF FIRST SUBARRAY *)
    L2, (* LARGEST OF SECOND SUBARRAY *)
    S2: (* SMALLEST OF SECOND SUBARRAY *)
    REAL;
BEGIN
    IF LAST < FIRST THEN WRITELN ('BAD DATA')
    ELSE
        IF FIRST = LAST
        THEN BEGIN (* CASE 1 *)
            SMALLEST := A[FIRST];
            LARGEST := A[FIRST]
        END
        ELSE IF LAST = FIRST + 1 (* CASE 2 *)
        THEN
            IF A[LAST] < A[FIRST]
            THEN BEGIN
                SMALLEST := A[LAST];
                LARGEST := A[FIRST]
            END
            ELSE BEGIN
                SMALLEST := A[FIRST];
                LARGEST := A[LAST]
            END
        ELSE BEGIN (* CASE 3 *)
            MIDDLE := (LAST - FIRST) DIV 2 +
                FIRST;

```



```

MAXMINR(FIRST, MIDDLE, L1, S1);
MAXMINR(MIDDLE+1, LAST, L2, S2);
IF L1 > L2
    THEN LARGEST := L1
    ELSE LARGEST := L2;
IF S1 < S2
    THEN SMALLEST := S1
    ELSE SMALLEST := S2
END
END;
```

This all looks very complicated. You might well be wondering whether it's worthwhile. To answer this question let us see what happens when we use this procedure to find the LARGEST and SMALLEST of an array A of 256 elements. As successive calls of MAXMINR take place, A is broken down as follows:

- (1) 2 arrays of 128 elements
- (2) 4 arrays of 64 elements
- (3) 8 arrays of 32 elements
- (4) 16 arrays of 16 elements
- (5) 32 arrays of 8 elements
- (6) 64 arrays of 4 elements
- (7) 128 arrays of 2 elements

Now let us count the number of comparisons of array elements that are required to get the final answer.

128 from final breakdown (7)	
64 to combine SMALLESTs for (6)	
64 to combine LARGESTs for (6)	
32 to combine SMALLESTs for (5)	
32 to combined LARGESTs for (5)	
.	
.	
.	
2 to combine SMALLESTs for (1)	
2 to combine LARGESTs for (1)	
380 comparisons	

Now if, as in the procedure at the beginning of this section, we simply run a FOR loop, doing two comparisons 255 times, we get a total number of comparisons of 510. Thus the alternative approach would save us about 25 percent of the work of the computation. For small arrays, this might not seem worthwhile. But it certainly is for large arrays!

The recursive procedure of Example 12-4 is very innocent-looking. Indeed, nothing we did not learn in previous chapters is used except that we allow the procedure to call itself. One could take the attitude that when a program calls itself all bookkeeping details will be taken care of and one need not be concerned with them. However, let us try to explain a little more about the details. Suppose we want to determine the largest and the smallest of the numbers $A[1] \dots A[8]$. According to the algorithm, we divide the numbers into two groups and find the largest and the smallest of the first group of numbers, $A[1] \dots A[4]$, then find the largest and the smallest of the second group of numbers, $A[5] \dots A[8]$. To find the largest and the smallest of the numbers $A[1] \dots A[4]$, we divide them again into two groups, $A[1], A[2]$ and $A[3], A[4]$. Similarly, we find the largest and the smallest of the numbers $A[5] \dots A[8]$ by dividing them into two groups. Finally, for groups of two numbers it is a trivial matter to find the largest and the smallest. The complete scheme is shown in Figure 12-4. Following the arrows from the left side to the right and back again, we see that Figure 12-4 shows clearly what happens when MAXMINR is called.

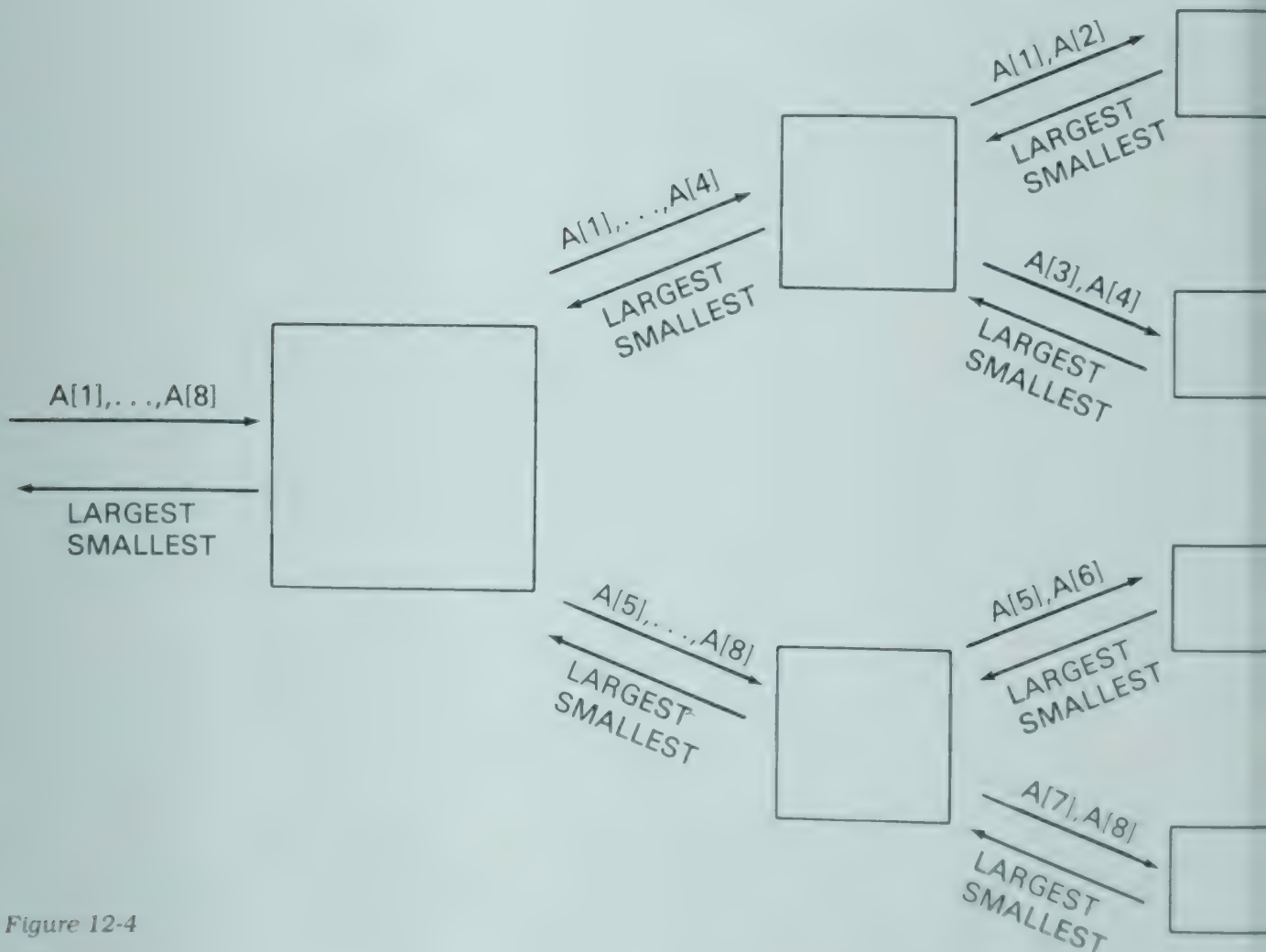


Figure 12-4

We stated in Chapter 11 that, in general, a procedure should be defined before its invocation. One might say that a recursive procedure is coming close to a violation of this rule, since it invokes itself within the definition of the procedure. Furthermore, there is a possibility that this general rule *must* be violated. For example, procedure A might call a procedure B, and then procedure B might need to call procedure A. Let us see how this might happen.

As an analogy of two procedures calling each other, suppose an electrical engineer is asked to design an electric power plant. Since he needs a computer for some control functions, he calls on a computer scientist to design a computer for him. After carrying out the preliminary design, the computer scientist finds that she needs some help with the electric circuits in the computer. She calls on the electrical engineer for this help. Upon receiving this help, she completes the design of the computer and sends it to the electrical engineer. The electrical engineer can now complete the design of the power plant.

In this example, we can imagine the electrical engineer and the computer scientist as two procedures. The electrical engineer is called upon to design a power plant. He in turn calls the computer scientist to design a computer. The computer scientist calls him to design some electric circuits. When the electric circuits are designed, they are returned to the computer scientist. The computer scientist can then complete the design of the computer and return it to the electrical engineer. The electrical engineer can then complete the design of the power plant.

In our discussion in Chapter 11, we said that procedure A can call procedure B, and procedure B can in turn call any procedure it wishes. A question we never raised in Chapter 11 was: Does it make any difference if procedure B calls a third procedure C or if procedure B calls procedure A? The situation is quite simple if A calls B and B calls C. When C completes its computation, the results are returned to B, and when B completes its computation, the results are returned to A, as shown in Figure 12-5.

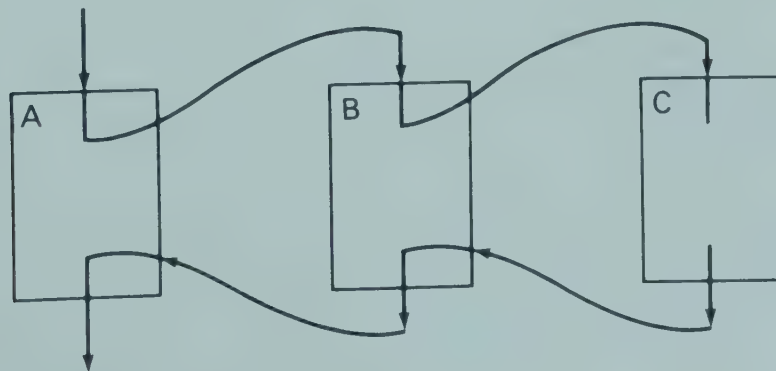


Figure 12-5

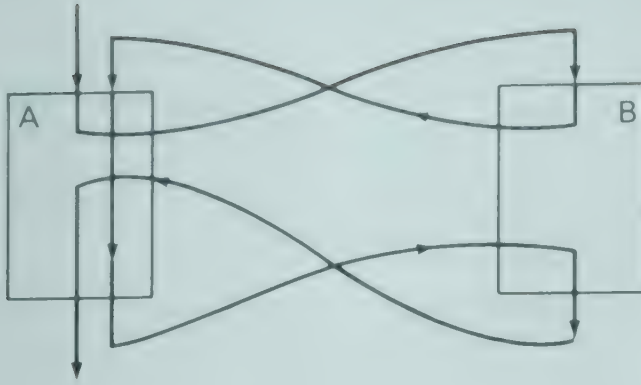


Figure 12-6

On the other hand, if A calls B and B calls A, the situation can be more complicated. Clearly, there is the possibility of a vicious circle as A calls B, B in turn calls A, A calls B again, B calls A again, and so on. On the other hand, a more constructive situation could be that A calls B, B calls A, A returns results to B, and B returns results to A to complete the computation. (This is what happened in the analogy of the electrical engineer and the computer scientist with which we began this section.) Figure 12-6 shows the successive calls and returns. Indeed, the possibility of two procedures calling each other is very much the same as that of a procedure calling itself recursively. We do not need to look too closely at details; it is sufficient to say that all bookkeeping details are automatically taken care of.

The only problem in writing procedures that call each other is to get around the rule that a procedure should be defined before it is called. But recall the reason for the rule. It was because otherwise the Pascal compiler wouldn't recognize the name in the procedure statement and would be unable to interpret that statement. What's really needed is a way to tell the compiler about procedures that will be defined later. This can be done by means of a *forward reference*. A forward reference looks like the heading of a procedure, complete with parameter list, followed by the word **FORWARD**. By putting a forward reference in a program before the procedure is invoked, we essentially declare the procedure so it can be used. Then when we later put the procedure itself in the program, the heading is abbreviated; specifically, the parameter list is omitted. Here's an outline of a program with a forward reference.

```
PROGRAM REF ( INPUT, OUTPUT );
VAR ...
PROCEDURE LATER ( M : INTEGER );
FORWARD;
PROCEDURE NOW ( N : INTEGER );
```

```

      .
      .
      .
    LATER (Q);
      .
      .
    END;
  PROCEDURE LATER;
  BEGIN
      .
      .
      .
    NOW (S);
      .
      .
    END;
  BEGIN (* MAIN PROGRAM *)
      .
      .
      .
    NOW (T)
      .
      .
  END.

```

12-5 SYNTAX

There is actually no new syntax introduced in this chapter. The provision for forward references is actually *outside* the formal specification of the syntax of Pascal. Thus you will notice that FORWARD is not included among the reserved words listed in Appendix A. It is a sort of special marker, indicating to the compiler that the block belonging to the preceding heading will be inserted later in the program. Accordingly, there is no formal syntax diagram for a forward reference.

Finally, as long as we are discussing syntax, recall that many of the syntax diagrams for Pascal are *recursive* in the sense that constructs are defined in terms of themselves. For example, the diagram for block includes block within it. This is precisely the same idea that is involved in recursive subprograms. Like the subprograms, the syntax definitions are not really circular, as they appear at first sight, since eventually we must reach a point where a simple definition (not recursive) holds. (In the case of a block, we cannot have blocks within blocks forever.) Just as recursive syntax diagrams make the Pascal language syntax easy to comprehend, so recursive subprograms are often the clearest way to define a computation.

12-6 NOTES ON STYLE AND PITFALLS

The most obvious pitfall in constructing a recursive subprogram is to forget that the recursive calls must eventually come to an end with a simple case that can be handled with an explicit calculation. This means that, in a sense, each successive call represents a slightly “simpler” case, until eventually the explicit calculation is reached.

Consider our example of Mr. Chan and his sons. If the difference between the ages of successive sons is 2, we might be tempted to compute AGEOFSON(N) as AGEOFSON(N+1) + 2, as follows:

```
FUNCTION AGEOFSON ( N : INTEGER );  
BEGIN  
    IF N = 1  
    THEN AGEOFSON := 27  
    ELSE AGEOFSON := AGEOFSON(N+1) + 2  
END;
```

Now if we call this function for $N = 3$, it will call itself for $N = 4$, which will cause a further call for $N = 5$ —and so forth, forever. This demonstrates that we don’t need a loop to cause an infinite program!

Another obvious pitfall in writing recursive subprograms is to forget to include code to handle all of the special, simple cases that the successive simple calls might work down to. As an exercise, study the procedure MAXMINR (Example 12-4) and determine what can happen if CASE 1 or CASE 2—or both—are omitted.

The use of functions or procedures as parameters in subprograms can also cause difficulties. Notice that only the *name* of the function or procedure is specified in the parameter list, and not its parameters and their types. This makes it difficult for the compiler to catch errors in the use of that function or procedure within the subprogram. The burden then falls on the programmer to make sure that the function or procedure is properly used within the subprogram.

12-7 A PROGRAMMING EXAMPLE: SORTING

In Chapter 9 we discussed one method for sorting an array of numbers. That algorithm (BUBBLESORT) involved exhaustive comparisons of adjacent pairs of numbers with swapping (interchange) of the members of any pair found to be out of order. In a way, this is analogous to finding the maximum (or minimum) value of an array of numbers by exhaustive comparisons. We found that we could solve the maximum problem more efficiently by subdividing the array, thus dividing the problem into two simpler problems. Similarly, we can take the approach of subdividing the array to produce a (usually) more efficient sorting algorithm. Since this approach means solv-

ing a problem by defining it in terms of solving two simpler problems, it is then naturally expressed in terms of recursive procedures. We shall discuss one such algorithm (named QUICKSORT) in this section.

Suppose that we have an array A of numbers that we want to sort into ascending order. Since we plan to proceed by breaking up the array into subarrays, we will assume that we are faced with the problem of sorting such a subarray, say $A[F]$, $A[F+1]$... $A[L]$. Now suppose that when we look at this set of numbers we discover that they are *partitioned*. This means that there is one element $A[J]$ in the list such that

1. $A[F]$, $A[F+1]$... $A[J-1]$ are all *less* than $A[J]$; and
2. $A[J+1]$, $A[J+2]$... $A[L]$ are all *greater* than $A[J]$.

(We will make the simplifying assumption in our discussion of QUICKSORT that there are no duplicate numbers.) Here's an example. Suppose the array contains the numbers:

4 1 2 5 10 7 15 12 8 9

Here we can use the number 5 to partition the array, since all numbers to the left are less than 5 and all numbers to the right are greater than 5. (Notice that there is nothing in the idea of partitioning that requires that the parts be of the same, or nearly the same, size.) There are two interesting points to notice about a partitioned array:

1. The element $A[J]$ is in its proper place in the *sorted* array. (Thus, in the example, 5 should eventually appear in the fourth slot in the array, as it does here.) This isn't magic. It's *because* all elements to the left are smaller and all elements to the right are larger.
2. The problem can now be broken down into two smaller problems:
 - a. Sort $A[F]$... $A[J-1]$.
 - b. Sort $A[J+1]$... $A[L]$.

In the context of the example, what this means is that once we sort 4 1 2 into 1 2 4 and 10 7 15 12 8 9 into 7 8 9 10 12 15, then, putting these pieces together, we have a complete sorted array.

You might well ask at this point whether one is likely to look at the array and find it partitioned in this convenient fashion. In fact, one is not! There are two parts involved in the QUICKSORT algorithm—one which takes an arbitrary array and turns it into partitioned form and one (the recursive part) that subdivides partitioned arrays and effects the actual sorting. We will postpone discussion of the former until later in this section. First we discuss the overall QUICKSORT procedure, written recursively. As usual, we will assume that the array A is declared globally.

QUICKSORT will need to call two other procedures—PARTITION (the partitioning procedure that we will discuss later) and SWAP (which performs

the basic operation of interchanging two given variable values). Here is a procedure SWAP:

```

PROCEDURE SWAP (VAR A, B : REAL);
VAR TEMP : REAL;
BEGIN
    TEMP := A;
    A := B;
    B := TEMP;
END;
```

We have discussed this procedure—and the need to declare TEMP a local variable—in the preceding chapter. SWAP clearly handles the case of sorting a two-element subarray whose elements are out of order. This is one of the “simple cases” which the recursive calls should work down to. What other special cases are needed? The only other case is when the subarray contains zero elements (or one), when nothing needs to be done. (A one-element array is already sorted.)

Here, then, is an encoding of the procedure QUICKSORT. We have included enough comments so that you should find it easy to follow. (To make it even easier, we have sacrificed some efficiency: specifically, we have not nested the IF statements.)

```

PROCEDURE QUICKSORT (F, L : INTEGER);
  (* PROCEDURE TO SORT A SEGMENT A[F]...A[L] *)
  (* OF A GLOBALLY DEFINED ARRAY A *)
  (* BY THE QUICKSORT ALGORITHM *)
VAR J : INTEGER;
BEGIN
  IF L <= F (* ARRAY HAS ZERO OR ONE ELEMENT *)
    THEN; (* DO NOTHING *)
  IF L = F + 1 (* ARRAY HAS 2 ELEMENTS *)
    THEN (* SWAP IF NECESSARY *)
      IF A[L] < A[F]
        THEN SWAP (A[F], A[L]);
  IF L > F + 1 (* MORE THAN 2 ELEMENTS *)
    THEN (* PARTITION AND SORT *)
      BEGIN
        PARTITION (F, L, J); (* RETURNS J *)
        QUICKSORT (F, J-1);
        QUICKSORT (J+1, L)
      END
END; (* QUICKSORT *)
```

The next thing we need to discuss is how the partitioning should be

done, i.e., what the procedure PARTITION should look like. First of all, here's what PARTITION must do for us:

1. Accept (as parameters) values of the first and last subscripts of the subarray that we want to have partitioned
2. Rearrange the subarray so that it is in partitioned form
3. Return to us (via a variable parameter) the subscript J of the partitioning element.

The question that immediately arises is how to determine J (or equivalently A[J]). Actually this choice is quite arbitrary, although the way the choice is made can affect the efficiency of the algorithm. For simplicity we will here choose A[F] (i.e., the first element of the subarray) as our standard of comparison. In the process of rearrangement, the value of A[F] will then be moved into the Jth spot in the subarray.

The next question is how to carry out the rearrangement. An efficient way to do this is to do some swapping—swapping “large” elements on the left side of the array with “small” elements on the right side. “Large” and “small” here mean “larger than A[F]” or “smaller than A[F].” To choose such pairs to be swapped, we begin at each end of the array (on the left with A[F + 1] and on the right with A[L] and move in toward the center. Here's how the scheme works for a numerical example.

Variable	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Value	4	9	6	1	2	7

↑
 Standard of comparison

Beginning on the *left* side (with A[2]) we see that $9 > 4$. Therefore 9 should eventually appear on the *right* side of the subarray, and we will want to swap it with something. Now beginning on the *right* side (with A[6]), we see that $7 > 4$, so A[6] need not be moved. However, moving toward the center, we find that $2 < 4$, so A[5] is out of place. Once we have found an element on each side that is out of place, we can swap them; that is, we swap A[2] and A[5], so our array now looks like:

Variable	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Value	4	2	6	1	9	7

Continuing to examine elements in this way, we find that 6 and 1 should be swapped, yielding

Variable	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Value	4	2	1	6	9	7

Now we are almost done. Elements 6, 9, and 7 on the right side are greater than 4, and elements 2 and 1 on the left side are less than 4. It remains to move 4 into its proper spot between the two parts. This can be accomplished by interchanging $A[1]$ with $A[3]$, giving us the following partitioned array:

Variable	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Value	1	2	4	6	9	7

The procedure PARTITION must carry out this rearrangement and return to the calling program the critical piece of information: the value of J , the subscript of the spot in the array where the standard of comparison has been put. (In the example, $J = 3$.) It appears that PARTITION must be very complicated, but actually this is a classic case of a subprogram that is simpler than the English-language description. The main problem is to keep track of where we are, i.e., which elements are being examined. This is easily done by introducing two variables: LEFT, to indicate the subscript of the variable being examined on the left side of the array, and RIGHT, to indicate the subscript of the variable being examined on the right side of the array. LEFT is incremented and RIGHT is decremented until we find elements $A[LEFT]$ and $A[RIGHT]$ to be swapped (or until $LEFT = RIGHT$, when we are done). Here is a Pascal version of PARTITION. (Again, more efficient code is possible.) Study this procedure carefully to convince yourself that it works as it should.

```

PROCEDURE PARTITION (F, L : INTEGER;
                     VAR J : INTEGER);
VAR LEFT, RIGHT : INTEGER;
    COMPARISON : REAL;
BEGIN
    (* INITIALIZATION *)
    COMPARISON := A[F];
    LEFT := F + 1;
    RIGHT := L;
    (* LOOK FOR FIRST ELEMENTS TO BE SWAPPED *)
    WHILE (LEFT <= L) AND (A[LEFT] < COMPARISON)
    DO LEFT := LEFT + 1;
    WHILE (RIGHT >= F + 1) AND
        (A[RIGHT] > COMPARISON)
    DO RIGHT := RIGHT - 1;
    WHILE LEFT < RIGHT
    DO BEGIN (* SWAPPING LOOP *)
        SWAP (A[LEFT], A[RIGHT]);
        LEFT := LEFT + 1;
        RIGHT := RIGHT - 1;
        WHILE A[LEFT] < COMPARISON

```

```

        DO LEFT := LEFT + 1;
    WHILE A[RIGHT] > COMPARISON
        DO RIGHT := RIGHT - 1
    END;
    (* INSERT STANDARD IN CORRECT POSITION *)
    SWAP (A[F],A[RIGHT]);
    (* RETURN PARTITION POINT *)
    J := RIGHT
END;

```

Notice that in this procedure we have set up a variable named COMPARISON to hold A[F], our standard of comparison. This is not only more efficient (avoiding repeated computation of the location of a subscripted variable) but also allows us to define the standard of comparison in some other way, if we wish, without necessitating a major rewrite of the procedure.

Finally we use the three procedures SWAP, QUICKSORT, and PARTITION in a complete program to read in and sort a set of N REAL numbers.

```

PROGRAM SORT(INPUT,OUTPUT);
VAR A: ARRAY[1..100] OF REAL;
    I, N: INTEGER;
PROCEDURE SWAP (VAR A,B: REAL);
    VAR TEMP: REAL;
    BEGIN
        TEMP := A;
        A := B;
        B := TEMP
    END;
PROCEDURE PARTITION (F, L: INTEGER;
                    VAR J: INTEGER);
    VAR LEFT, RIGHT: INTEGER;
        COMPARISON: REAL;
    BEGIN
        COMPARISON := A[F];
        LEFT := F+1;
        RIGHT := L;
        WHILE (LEFT <= L) AND (A[LEFT] < COMPARISON)
            DO LEFT := LEFT + 1;
        WHILE (RIGHT >= F+1) AND
            (A[RIGHT] > COMPARISON)
            DO RIGHT := RIGHT - 1;
        WHILE LEFT < RIGHT
            DO BEGIN
                SWAP (A[LEFT], A[RIGHT]);
                LEFT := LEFT + 1;
                RIGHT := RIGHT - 1;
                WHILE A[LEFT] < COMPARISON

```

```

        DO LEFT := LEFT + 1;
        WHILE A[RIGHT] > COMPARISON
            DO RIGHT := RIGHT - 1
        END;
        SWAP (A[F], A[RIGHT]);
        J := RIGHT
    END;
    PROCEDURE QUICKSORT (F, L: INTEGER);
        (* PROCEDURE TO SORT A SEGMENT A[F]..A[L] *)
        (* OF A GLOBALLY DEFINED ARRAY A BY THE *)
        (* QUICKSORT ALGORITHM *)
    VAR J: INTEGER;
    BEGIN
        IF L <= F (* ARRAY HAS ZERO OR ONE ELEMENT *)
            THEN; (* DO NOTHING *)
        IF L = F + 1 (* ARRAY HAS TWO ELEMENTS *)
            THEN (* SWAP IF NECESSARY *)
                IF A[L] < A[F]
                    THEN SWAP (A[F], A[L]);
        IF L > F+1 (* MORE THAN TWO ELEMENTS *)
            THEN (* PARTITION AND SORT *)
                BEGIN
                    PARTITION(F,L,J); (* RETURNS J *)
                    QUICKSORT(F,J-1);
                    QUICKSORT(J+1,L)
                END
        END;
    BEGIN (* MAIN PROGRAM *)
        READ(N);
        FOR I := 1 TO N DO
            READ(A[I]);
        QUICKSORT(1,N);
        FOR I := 1 TO N DO
            WRITELN (A[I])
        END.

```

12-8 EXERCISES

12-1. What is printed by the following?

```

PROGRAM T(INPUT,OUTPUT);
    FUNCTION F(Y:INTEGER):INTEGER;
    BEGIN
        IF Y <= 0 THEN F:=2
            ELSE F:=F(Y-1)*F(Y-2)
        END;

```



```

BEGIN
    WRITELN ( F ( 2 ) )
END .

```

- 12-2.** What is printed when the following function is called by the statement WRITELN(A(1,2))?

```

FUNCTION A(M,N: INTEGER): INTEGER;
VAR X: INTEGER;
BEGIN
    IF M=0
    THEN X:= N+1
    ELSE IF N=0
        THEN X:=A(M-1,1)
        ELSE X:=A(M-1,A(M,N-1));
    WRITELN('A(' ,M, ',' ,N, ')=' ,X);
    A:=X
END;

```

- 12-3.** The following function was written to determine the larger of two integers.

```

FUNCTION MAX ( I, J : INTEGER ) : INTEGER;
BEGIN
    MAX := I;
    IF J > MAX THEN MAX := J
END;

```

However, when it was run strange things happened. Why? (Try it on your system.)

- 12-4.** Pascal's triangle appears below:

```

Row
  1           1
  2         1   1
  3       1   2   1
  4     1   3   3   1
  5   1   4   6   4   1

```

The value of any particular entry in Pascal's triangle is the sum of the two numbers which are immediately above and to the left and right, respectively, of that number. Thus 4 in the triangle above is calculated by adding 1 and 3 in the row above.

Write a recursive procedure PASCAL that has three parameters: I, J, and K. PASCAL must be called with arguments for parameters I and J. PASCAL computes the value K, which it will return. K is the value of the Jth entry in the Ith row of Pascal's triangle. For example, calling PASCAL (4,2,X) will give X the value 3. Calling PASCAL (5,3,X) will give X the value 6. Note that PASCAL (Y,1,X) and PASCAL (Y,Y,X) return 1 as the value of X for any value of Y.

- 12-5.** $N!$ (factorial N) is defined to be $N * (N - 1) * (N - 2) \dots * 2 * 1$.
- (a) Write a recursive function to compute $N!$.
 - (b) Write a nonrecursive function to compute $N!$.
 - (c) Which function is "better" and why? What do you mean by "better"?
- 12-6.** Write a recursive procedure to find the GCD (greatest common divisor) of a pair of numbers using Euclid's algorithm. Euclid's algorithm may be expressed as follows:

```
DO WHILE A is not equal to B:
    Replace the larger of A
    and B by the positive
    difference between them.
END;
```

(When A equals B , the value of the GCD is A .)

- 12-7.** Write a procedure that, given $N \geq 0$, will print the number of ways that a frog can climb N stairs. The frog can climb either one, two, or three steps at a leap. For example, if $N = 3$, the stairs can be climbed 1-1-1 or 1-2 or 2-1 or 3, so your procedure would print "4."
- 12-8.** Write a function to compute

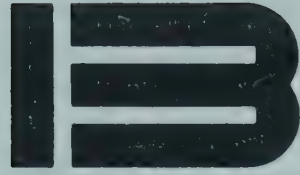
$$f(N) = \begin{cases} 1 & \text{if } N \leq 1 \\ f(N-1) + 2f(N-2) & \text{if } N > 1 \end{cases}$$

for integers N . How many calls to the function are executed if it is initially called with $N = 6$? What is the value of $f(6)$?

SUMMARY OF CHAPTER 12

A subprogram (function or procedure) that calls itself is known as a *recursive subprogram*. In Pascal, any subprogram may call itself, and whatever bookkeeping that is necessary is provided automatically by the compiler. (In some programming languages, such as FORTRAN, all subprograms are nonrecursive; that is, a subprogram is not allowed to call itself. In other programming languages, such as PL/I, recursive subprograms must be declared explicitly as such.)

In Pascal it is also possible for two subprograms to call each other. However, in that case a forward reference declaration is necessary.



SIMPLE DATA TYPES

13-1 INTRODUCTION

So far you have learned to use four *simple* types of constants and variables, namely, INTEGER, REAL, BOOLEAN, and CHAR. Called “simple” because any variable or constant of those types can only have a value that is a single number or character, they are also sometimes referred to as “scalar” types, or “unstructured” types. You have also learned about one structured (not simple) type, namely, the array. What distinguishes structured from unstructured types is that structured types are built up from individual components that are simple. (Thus you can declare ARRAY OF CHAR, ARRAY OF INTEGER, etc.) In this chapter we will look closely at simple types and explain how you can define your own. In later chapters we will look at some other useful structured types.

13-2 DEFINING NEW SIMPLE TYPES BY ENUMERATION

Suppose we hire a programmer from Mars who tells us that, instead of writing 1, 2, 3, 4, Martians write $-$, $=$, $\equiv$, $\square$. Consequently, she would prefer to write statements like

I := -
J := $\square$

in her programs. In Pascal, she can almost do this. Our computer can't handle these funny looking characters: but if it could, all our Martian programmer would have to do is to define a new data type as follows:

```
TYPE MARS = ( -, =, ≡, □ );
```

This defines a new data type called MARS that has four possible values: `-`, `=`, `≡`, and `□`. Such type definitions are inserted in a program or procedure just after the `CONST` definitions and before the `VAR` declarations. Then, after defining the new type, our programmer can define variables of that type in the usual way:

```
VAR I, J : MARS;
```

Now the assignment statements above are legal.

Our Martian programmer has taken advantage of the friendly Pascal facility that allows us to define new simple types by enumeration; that is, by listing all the values that a variable of that type can take on. (Such types are referred to as *enumerated types*.) This facility is very useful even if we aren't from Mars. For example, suppose we wish to write a program to keep track of the electric power consumption in a week. Although we could refer to the seven days in the weeks as day 1, 2, ..., 7, it will be more convenient to refer to them explicitly as MONDAY, TUESDAY, ..., SUNDAY. This can be done if we declare a new data type, which we shall call DAY.

```
TYPE DAY = ( MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
              FRIDAY, SATURDAY, SUNDAY );
```

As you can see by the absence of quotation marks, the MONDAY, ..., SUNDAY in this declaration are *not* to be confused with character strings. Although they follow the rules for Pascal identifiers, they are the constant values that variables of the defined type may take on. Once we have declared the new data type DAY, we can declare variables whose values are those of the data type DAY. Thus, we may have

```
VAR LOW, PEAK : DAY;
```

to declare LOW, PEAK to be variables of the type DAY. In other words, the variables LOW, PEAK can assume any one of the values MONDAY, TUESDAY, ..., SUNDAY. Consequently, we may have assignment statements such as

```
LOW := WEDNESDAY;  
PEAK := MONDAY
```

You can see that, as far as setting up variables and assigning values to them is concerned, there is no conceptual difference between this new data type and the standard simple types. But you cannot do everything with

enumerated types that you can do with standard types. For example, you cannot write (print out) values of enumerated types, nor can you read them or use them in arithmetic expressions. (In the latter two respects, of course, they are like the type `BOOLEAN`.) However, you can use enumerated types in conditional expressions and `CASE` statements, as will be discussed in the next sections.

13-3 ORDERING OF VALUES

One of the characterizing features of simple data types is that their values are always ordered. In other words, we can always compare two distinct values of the same data type and determine the *smaller* and the *larger* of the two. For the types `INTEGER` and `REAL`, such an ordering is the natural numerical ordering. Thus, we have

$$\begin{aligned} 3 &< 7 \\ 5.2 &< 100.8 \end{aligned}$$

For the type `BOOLEAN`, it is understood that `FALSE` < `TRUE`. For the type `CHAR`, as you may recall, the ordering is implementation-dependent. However, it is always true that

$$'0' < '1' < '2' < \dots < '9'$$

and

$$'A' < 'B' < 'C' \dots < 'Z'$$

For any enumerated data type, the ordering is defined by the order in which the values are listed in the definition. Thus, for example, for the data type `COLOR` defined as

```
TYPE COLOR = (WHITE, BLACK, YELLOW, GREEN);
```

we have `WHITE` < `BLACK` < `YELLOW` < `GREEN`.

Then we can use enumerated types in any kind of statement that includes a conditional expression. For example, we might declare a variable `PLANTCOLOR` of type `COLOR` and write

```
IF PLANTCOLOR < GREEN THEN
  WRITELN ('THIS PLANT IS SICK')
```

Or we might define a variable of type `COLOR` named `HAT` and write

```
IF HAT = WHITE THEN
  WRITELN ('HE'S A GOOD GUY.')
```

```
ELSE IF HAT <> BLACK THEN
  WRITELN ('HE MIGHT BE OK.')
```

Similarly we can use enumerated types in REPEAT or WHILE statements:

```
WHILE PLANTCOLOR <> GREEN DO
  WATERPLANT
```

and even in FOR loops:

```
I := 1;
FOR HAT := WHITE TO GREEN DO
BEGIN
  I := I + 1;
  WRITELN (NAME [I])
END
```

This last example shows a usage which is legal for the standard simple types INTEGER, CHAR, and BOOLEAN, but not for type REAL. The first three types are sometimes referred to as *ordinal types* because attached to each value there is an integer (called the *ordinal number*) that gives its order. For the type INTEGER, this integer is the value itself. For the type CHAR the ordinal numbers of the values are defined by the implementation. (See Section 11-11.) For the type BOOLEAN, the ordinal number of FALSE is 0 and that of TRUE is 1 (hence FALSE < TRUE). Any enumerated type that you define is also an ordinal type; the associated integers are 0, 1, 2, 3,... attached to the values in the order in which you list them.

As we explained in Chapter 11, Pascal has the standard functions ORD, SUCC, and PRED that allow us to look at (and hence work with) the ordinal number of each ordinal-type value and to determine the successor or predecessor of each value in the sequence. Thus we find that, if a variable NUM has INTEGER value -5 and a variable BOOL has BOOLEAN value FALSE, then ORD(NUM) is -5, ORD(BOOL) is 0, SUCC(NUM) is -4, and SUCC(BOOL) is TRUE. These functions can then also be used with enumerated types. Thus for the type COLOR defined above we find that

```
ORD (BLACK) = 1
SUCC (YELLOW) = GREEN
PRED (BLACK) = WHITE
```

The computer has no problem, then, carrying out a FOR loop:

```
FOR HAT := WHITE TO GREEN DO
```

since this has the same effect as defining a control variable—corresponding to ORD(HAT)—that takes on values from ORD(WHITE) to ORD(GREEN), i.e., from 0 to 3.

13-4 USING ENUMERATED TYPES IN CASE STATEMENTS

We hope that you have been making use of CASE statements, using case indexes that evaluate to INTEGER or CHAR values to determine which of several statements to execute. Enumerated types are particularly useful in CASE statements; they tend to make the different choices—the case values—much more intelligible to the reader than if integer values are used.

Suppose, for example, we want to set the maximum and minimum number of semester hours that a college student should take each semester, and that these numbers depend on whether the student is a freshman, sophomore, junior or senior. We might then define a type

```
TYPE YEAR = (FRESHMAN, SOPHOMORE, JUNIOR,  
             SENIOR);
```

and then declare variables

```
VAR JOHN, MARY : YEAR;
```

which will be assigned values specifying the classes of these students. Then the following CASE statement can be used to set the values of MAX and MIN.

```
CASE JOHN OF  
  FRESHMAN : BEGIN  
    MIN = 14;  
    MAX = 18  
  END;  
  SOPHOMORE : BEGIN  
    MIN = 16;  
    MAX = 20  
  END;  
  JUNIOR : BEGIN  
    MIN = 18;  
    MAX = 22  
  END;  
  SENIOR : BEGIN  
    MIN = 18;  
    MAX = 22  
  END  
END
```

As another example, suppose that a hotel charges different rates depending on whether the night of the stay was a holiday, weekend, or weekday. Then we might define

```
TYPE DAY = (HOLIDAY, WEEKEND, WEEKDAY);
```

and declare a variable

```
VAR NIGHT : DAY;
```

Then we can assign the rates by a CASE statement:

```
CASE NIGHT OF
  HOLIDAY : RATE := 20;
  WEEKEND : RATE := 25;
  WEEKDAY : RATE := 30
END
```

Or if we want to set a base rate and adjust it for the special cases, we remember that we can include a null statement and write:

```
RATE := 30;
CASE NIGHT OF
  HOLIDAY : RATE := RATE - 10;
  WEEKEND : RATE := RATE - 5;
  WEEKDAY:
END
```

We shall end this section by expanding this example into a complete program. Example 13-1 reads in a code (an integer) giving the type of day of a hotel stay, converts the code to the appropriate value of NIGHT, sets the room rate with a CASE statement, then reads the number of nights stayed and computes and prints the bill. We could, it appears, have saved ourselves some trouble by computing the room rate directly from the code. But by using the enumerated type we ensure that a readable description of the room rate structure and the meanings of the codes are built into the program.

EXAMPLE 13-1

```
PROGRAM HOTEL (INPUT, OUTPUT);
TYPE DAY = (HOLIDAY, WEEKEND, WEEKDAY);
VAR NIGHT : DAY; NUMNIGHTS, CODE: INTEGER;
    RATE, BILL: REAL;
BEGIN (* HOTEL *)
  READ(CODE);
  WHILE CODE <> 0 DO (* = 0 MEANS END OF DATA *)
  BEGIN (* WHILE LOOP *)
    CASE CODE OF
      1: NIGHT := HOLIDAY;
      2: NIGHT := WEEKEND;
      3: NIGHT := WEEKDAY
    END;
```

```

CASE NIGHT OF
    HOLIDAY: RATE := 20;
    WEEKEND: RATE := 25;
    WEEKDAY: RATE := 30
END;
READLN (NUMNIGHTS);
BILL := RATE * NUMNIGHTS;
WRITELN ('FOR', NUMNIGHTS:3,
        ' NIGHTS, YOU PAY', BILL:7:2);
READ (CODE)
END (* WHILE *)
END. (* HOTEL *)

```

13-5 SUBRANGE TYPES

Suppose we want to write a program to compile statistics on student enrollments in the first hundred years of the history of the University of Illinois. We can use an integer variable YEAR to denote the year. In that case, the value of YEAR ranges from 1868 through 1967. If whenever the program is executed the value of YEAR is less than 1868 or is larger than 1967, we know that there must be an error. In Pascal, such an error will be detected if we not only declare the variable YEAR to be an integer, but also declare YEAR to be an integer whose value lies within the range 1868 through 1967. To this end, we define a new data type that is an INTEGER data type, but with the restriction that the value of any variable of this data type must be in the range 1868 through 1967. Thus, we write

```

TYPE INDEX = 1868..1967;
VAR YEAR, ENROLL : INDEX;

```

In this way, we define a new data type called INDEX and declare that YEAR and ENROLL are variables of this type.

A data type that assumes only values within a subrange of the values of another data type is called a *subrange data type*. Subrange data types can be defined for any ordinal data type. As was pointed out in the preceding section, for these data types there is an ordering over all possible values. Consequently, a subrange type can be defined by specifying the lower limit of the subrange and the upper limit of the subrange. It is not necessary to explicitly specify the type upon which a subrange type is based (the *associated type*), since the compiler will recognize the type of the values that specify the limits of the subrange. Thus, since 1868 and 1967 are integers, INDEX is clearly a subrange of the type INTEGER. Similarly,

```

TYPE NOTE = 'A'..'G';

```


defines a subrange of type CHAR. And if we have defined

```
TYPE COLOR = (RED, BLUE, WHITE, GREEN, ORANGE,
              BROWN);
```

then

```
TYPE PATRIOT = RED..WHITE;
```

defines a subrange of the type COLOR. The operations that may be carried out on a subrange type are, of course, just those that are legal for its associated type.

Incidentally, in declaring variables of subrange types, we can often use a shortcut and omit the explicit type declaration. Thus, instead of defining the type INDEX we could simply have declared

```
VAR YEAR, ENROLL : 1868..1967;
```

But if we wanted to declare an array of type INDEX, we would have needed to predefine that type.

We should note that it makes no sense to define a subrange of the type BOOLEAN. And we emphasize that, since REAL is *not* an ordinal type, no subrange of type REAL can be defined.

As a final word in this section, we note that both enumerated and subrange types may be used as index types in array declarations. For example, to set up an array to hold enrollment data for the years 1868 through 1967, we could declare

```
TYPE INDEX = 1868..1967;
VAR ENROLL : ARRAY [INDEX] OF INTEGER;
```

instead of

```
VAR ENROLL : ARRAY [1868..1967] OF INTEGER;
```

Similarly, to set up an array to hold the number of cans of various colored paints on hand, we could declare

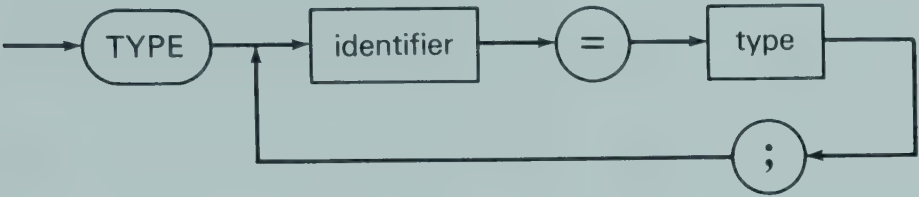
```
TYPE COLOR = (WHITE, RED, GREEN, BLUE, BROWN);
VAR INVENTORY : ARRAY [COLOR] OF INTEGER;
```

Then, for example, we could assign

```
INVENTORY [BROWN] := 10
```

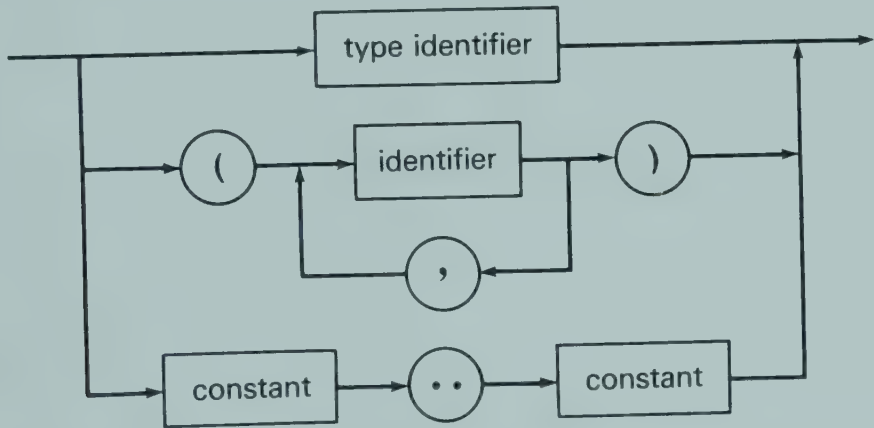
if there are 10 cans of brown paint.

The new syntax described in this chapter is that needed for defining new data types—enumerated and subrange. The diagram for a *type definition* is:



As we mentioned above, type definitions are inserted between the constant definitions and the variable declarations in your program. (See diagram for block in Appendix B.)

We have not yet said what the box labeled “type” can consist of. This chapter has only dealt with the simple (or unstructured) types. The structured types and how they are defined will be discussed in later chapters. Here is the diagram for *simple type*.



On the first line of this diagram, the “type identifier” may be the identifier of any other defined or standard type. You will recognize the second and third lines as the syntax for defining enumerated and subrange types, respectively.

**13-7 A PROGRAMMING
EXAMPLE: ANALYZING
STUDENT GRADES**

Grade cards have been typed for a group of N students ($N \leq 1000$) in a programming course. Each card contains the student's ID (an integer from 1 to 1000) and grade ('A', 'B', 'C', 'D', or 'E') for the course. We'd like to know how well the students from the different classes (freshman, sophomore, junior,

and senior) have performed. Luckily, we are able to get the university records office to generate another set of cards containing student ID and class (coded as 1, 2, 3, and 4). Using this data, we plan to generate a table giving the percent of each class making A's, B's, etc. The table should look something like the following:

	Percent of Students Receiving				
	A	B	C	D	E
Freshman	5	10	30	40	15
Sophomore	10	20	30	30	10
Junior	15	25	30	15	15
Senior	30	20	10	20	20

(Notice that each row should sum to 100.)

Since the only link between a grade and the class of the student that received that grade is through the ID, we shall begin by reading the grade and class data into two arrays, both indexed by ID. (Assume that each set of cards ends with a dummy card containing an ID of zero.) Then we shall scan these arrays, accumulating the numbers of each grade made by members of each class in a two-dimensional array. It will then be an easy task to compute the desired percentages and print the table.

The programming task is quite simple. But we can use it to illustrate the use of the new types introduced in this chapter. The class codes, IDs, and grades are all best defined as subrange types. Furthermore we can make the program more readable by internally using a variable named CLASS with values FR, SO, JR, and SR defined as an enumerated type. And we'll add an extra value of NUL so that, when we scan the arrays, we can recognize when no class has been entered for a particular ID. Notice that then we can provide meaningful index values for the two-dimensional array GRADECOUNT. Specifically, we use FR..SR for the row index and 'A'..'E' for the column index.

EXAMPLE 13-2

```

PROGRAM ANALYSIS (INPUT,OUTPUT);
TYPE CLASS = (FR, SO, JR, SR, NUL);
      GRADE = 'A'..'E';
VAR ID: 0..1000;
    TOTAL: INTEGER;
    CODE: 1..4;
    G: GRADE; C: CLASS;
    GRADES: ARRAY[1..1000] OF GRADE;
    CLASSES: ARRAY[1..1000] OF CLASS;
    GRADECOUNT: ARRAY[FR..SR, 'A'..'E'] OF
      INTEGER;

```


BEGIN

```
(* SET CLASSES TO NUL; GRADECOUNT TO ZERO *)
FOR ID := 1 TO 1000 DO
  CLASSES[ID] := NUL;
FOR G := 'A' TO 'E' DO
  FOR C := FR TO SR DO
    GRADECOUNT[C,G] := 0;
(* PRINT TABLE HEADINGS *)
WRITELN ('PERCENT OF STUDENTS RECEIVING': 45);
WRITELN ('A': 16, 'B': 10, 'C': 10, 'D': 10,
        'E': 10);
(* LOAD GRADES *)
READ (ID);
WHILE ID>0 DO
  BEGIN
    READ (GRADES[ID]);
    READ (ID)
  END; READLN;
(* LOAD CLASSES *)
READ (ID);
WHILE ID>0 DO
  BEGIN
    READ (CODE) ;
    CASE CODE OF
      1: CLASSES[ID] := FR;
      2: CLASSES[ID] := SO;
      3: CLASSES[ID] := JR;
      4: CLASSES[ID] := SR
    END;
    READ (ID)
  END; READLN;
(* ACCUMULATE GRADECOUNT ENTRIES *)
FOR ID := 1 TO 1000 DO
  IF CLASSES[ID] <> NUL THEN
    GRADECOUNT[CLASSES[ID], GRADES[ID]] :=
      GRADECOUNT[CLASSES[ID],GRADES[ID]]+1;
(* COMPUTE AND PRINT PERCENTAGES *)
FOR C := FR TO SR DO
  BEGIN
    TOTAL := 0;
    FOR G := 'A' TO 'E' DO
      TOTAL := TOTAL + GRADECOUNT[C,G];
    CASE C OF
      FR: WRITE ('FRESHMAN ');
      SO: WRITE ('SOPHOMORE ');
      JR: WRITE ('JUNIOR ');
      SR: WRITE ('SENIOR ')
    
```

```

END;
FOR G := 'A' TO 'E' DO
  WRITE ((GRADECOUNT[C,G]/TOTAL)*100.0:
    10:2);
  WRITELN
END
END.

```

To test this program, we ran it with the following data file:

```

1A
2B
3C
4D
5E
6A
7B
8C
9D
10E
11A
0
1 1
2 3
3 2
4 1
5 4
6 2
7 3
8 1
9 2
10 4
11 4
0

```

And the output was:

	PERCENT OF STUDENTS RECEIVING				
	A	B	C	D	E
FRESHMAN	33.33	0.00	33.33	33.33	0.00
SOPHOMORE	33.33	0.00	33.33	33.33	0.00
JUNIOR	0.00	100.00	0.00	0.00	0.00
SENIOR	33.33	0.00	0.00	0.00	66.67

(Actual output of the program spaces columns differently)

13-8 EXERCISES

13-1. What is the output of the following program?

```
PROGRAM EX131 (INPUT,OUTPUT);
TYPE COLOR = (RED,WHITE,BLUE,GREEN,PURPLE,BLACK);
VAR FLAG: COLOR;
    RAINBOW : BLUE..PURPLE;
BEGIN
    FLAG := SUCC(RED);
    RAINBOW := PRED(PURPLE);
    WRITELN (ORD(FLAG) + ORD(RAINBOW));
    FOR FLAG := RED TO BLACK DO
        WRITELN (ORD(FLAG))
    END.
```

13-2. What is printed by the following?

```
PROGRAM EX2 (INPUT,OUTPUT);
TYPE COLOR = (RED,WHITE,BLUE,GREEN,PURPLE,BLACK);
VAR FLOWER,SKY: COLOR;
BEGIN
    IF ORD(SUCC (RED))<>2 THEN
        BEGIN
            FLOWER := RED;
            IF SUCC(BLUE) = GREEN THEN
                SKY := BLUE
            ELSE FLOWER := WHITE
            END
        ELSE BEGIN
            SKY := WHITE;
            FLOWER := PRED(BLACK)
            END;
        CASE SKY OF
            BLUE: WRITELN ('SKY IS BLUE. ');
            WHITE: WRITELN ('SKY IS WHITE. ');
            END;
        WRITE ('FLOWER IS ');
        CASE FLOWER OF
            RED: WRITELN ('RED. ');
            WHITE: WRITELN ('WHITE. ');
            PURPLE: WRITELN ('PURPLE. ')
            END
    END.
```


- 13-3.** You will recall from Chapter 11 that there is a standard function `CHR` that will give you the character corresponding to a specified ordinal number. For example, `CHR(1)` returns the value 'A'. You may find this kind of function useful for enumerated types. For example, given the definition

```
TYPE COLOR = (RED, WHITE, BLUE, GREEN, PURPLE, BLACK);
```

write a function `COLR` that, given an ordinal number between 0 and 5, returns one of these color values. Be sure to provide for an error message in case the given ordinal number is out of range.

Test your function by using it in a program that reads two integers `N` and `M` and prints a list of colors with ordinal numbers between `N` and `M` (or an appropriate message if there are none).

- 13-4.** In Example 13-1, only a limited range of values is meaningful for the variable `CODE`. Rewrite this example using a subrange type for `CODE`, so that mistyped codes will be caught on input.
- 13-5.** If the user of a program has to remember code values, the program becomes less useful. Rewrite Example 13-1 as an interactive program that prompts the user for `NUMNIGHTS` and `CODE`. The prompt message should explain the meanings of the code values. The program should display an error message and give the user another chance if an invalid code is typed.
- 13-6.** Look back at the Pascal programs you have written to date. Where could enumerated or subrange types profitably have been used?

SUMMARY OF CHAPTER 13

New data types can be defined by listing (enumerating) all possible values of each type. Such types are called *enumerated types*. Furthermore, the values of an enumerated type are ordered (for comparison purposes) according to the order in which they are enumerated.

Based on a given ordinal data type (that is, a type in which the values are ordered), we can define a subrange data type that has values only within a subrange of the given data type. The definition of a subrange data type not only makes some programming steps slightly simpler, but also provides a way to prevent inadvertent use of values outside a prescribed subrange. (Such usage will cause a type error.)

SETS AND ARRAYS AS STRUCTURED TYPES

14-1 INTRODUCTION

Consider the problem of writing a program to keep track of the courses a student takes. In the program, we shall use information on the courses offered by the department of computer science, the courses a student registered for at the beginning of the semester, the courses the student passed, the courses required for graduation, and so on. We recall from mathematics that a convenient notation in which to represent this kind of information is set notation. Thus, we might define the following sets to represent the courses offered by the department of computer science, the courses a student registered for, and the courses required for graduation, respectively:

$$\begin{aligned}\text{COURSES} &= \{\text{CS121, CS221, CS257, CS264, CS273,} \\ &\quad \text{CS281, CS321, CS323, CS325}\} \\ \text{REGIS} &= \{\text{CS257, CS264, CS321, CS325, PE107,} \\ &\quad \text{ENG105}\} \\ \text{REQUIRED} &= \{\text{CS121, CS221, CS257, CS264, CS273}\}\end{aligned}$$

There are many questions one might wish to ask. For example: Is the student taking some courses outside the department? What elective courses in computer science is he or she taking? We see that we can answer these questions by examining the sets we have defined.

As another example of the use of sets, consider the management of an automated library system in which users can request information about books they are looking for. In such a system, a book can be described by the

attributes it possesses. For example, the table below shows the attributes of the four books: A, B, C, and D.

	Fiction	Published in 1981	Multiple copies	On reserve	Paperback	Copy available
A	X		X		X	
B		X		X		
C		X	X	X		
D	X			X		X

Set notation can be used to represent all the possible attributes a book might possess, as follows:

ATTRIBUTE = {FICTION, 1981, MULTIPLE, RESERVED,
PAPERBACK, AVAILABLE}

Then the attributes of each book are subsets of the set ATTRIBUTE:

A = {FICTION, MULTIPLE, PAPERBACK}
 B = {1981, RESERVED}
 C = {1981, MULTIPLE, RESERVED}
 D = {FICTION, RESERVED, AVAILABLE}

From these sets we will be able to answer questions such as: Is book A a paperback published in 1981? Are books C and D both fiction? Is either book A or book D available?

14-2 ELEMENTARY SET THEORY

In this section we review briefly some of the basic concepts and terminology of elementary set theory.

A *set* is a collection of distinct objects. Thus, the group of courses offered by the department of computer science is a set. So is the group of courses a student has registered for, and so is the group of courses required for graduation. We use the notation $\{a, b, c\}$ to denote the set which is the collection of the objects a , b , and c . The objects which make up a set are sometimes called its *elements*.

We also give names to sets. Thus, for example, we write

REQUIRED = {CS121, CS221, CS257, CS264, CS321}

to mean that the set named REQUIRED is the collection of the five objects (or elements) CS121, CS221, CS257, CS264, and CS321. We can then say that

CS121 is an object in the set REQUIRED, and also that the set REQUIRED contains the object CS121. Similarly, we say that the set REQUIRED does not contain the object CS273. Alternatively, we can say that CS121 is a *member* of the set REQUIRED, while CS273 is not. An *empty set* is a set that contains no members; it is denoted by $\{ \}$. For example, let FAIL denote the set of courses that a student fails. If the student passes all his or her courses, then FAIL is an empty set and we can write $\text{FAIL} = \{ \}$.

Given two sets REGIS and COURSES, we say that REGIS is a subset of COURSES if every element of REGIS is also an element of COURSES. The set REGIS is then also said to be *included* in the set COURSES; and COURSES is said to *include* the set REGIS. Two sets P and Q are said to be *equal* if P is a subset of Q and Q is also a subset of P. In other words, P and Q contain the same collection of objects.

One can also talk about *proper* subsets to indicate the case where one set is a subset of, but is not equal to, another. Thus REQUIRED is a proper subset of COURSES.

The *union* of two sets is the set whose elements are exactly those in either or both of the two sets. The *intersection* of two sets is the set whose elements are exactly those in both of the two sets. The difference $(P - Q)$ of two sets P and Q is the set whose elements are exactly those that are in P but not in Q. For example, if

$$\begin{aligned} P &= \{\text{FICTION, MULTIPLE, PAPERBACK}\} \\ Q &= \{\text{FICTION, 1981, PAPERBACK}\} \end{aligned}$$

Then the union of P and Q is the set {FICTION, MULTIPLE, 1981, PAPERBACK}; the intersection of P and Q is the set {FICTION, PAPERBACK}; and the difference $P - Q$ is the set {MULTIPLE}.

14-3 SETS IN PASCAL

For many scalar data types, we can define a new data type that is a set of elements of the scalar type. In other words, a variable of the new data type will have as its value a set of values of the given scalar data type, which is referred to as the *base type*. For example, suppose that we have defined

```
TYPE COLORS = (RED, BLUE, YELLOW, WHITE);
```

Then we can define a new type

```
TYPE SETCOLOR = SET OF COLORS;
```

A variable of type SETCOLOR will have as its value some set of colors. For example, after declaring

```
VAR X, Y, Z: SETCOLOR;
```

(i.e., after declaring X, Y, and Z to be variables of type SETCOLOR) we can assign set values to them. In Pascal, we specify a set as usual by listing its elements. But the list is enclosed in square brackets instead of the curly brackets used in set theory. Thus we might write:

```
X := [RED, YELLOW];
Y := [WHITE];
Z := [BLUE, YELLOW]
```

These statements assign the value [RED, YELLOW] (or {RED, YELLOW} in the usual set notation) to X, the value [WHITE] to Y, and the value [BLUE, YELLOW] to Z. Note that you must be careful to distinguish between [WHITE], a set containing the single object WHITE, and the object WHITE itself. If you write

```
Y := WHITE
```

you will get a type error, since Y must be a *set* of colors.

In this same way, we can create the sets we need for the examples in Section 14-1. For example, we can define the two data types COURSES and COLLECTIONS by

```
TYPE COURSES = (CS121, CS221, CS257, CS264,
                 CS273, CS281, CS321, CS323,
                 CS325);
COLLECTIONS = SET OF COURSES;
```

We can now write

```
VAR   EASIEST, HARDEST : COURSES;
      REGIS, REQUIRED : COLLECTIONS;
BEGIN
  EASIEST := CS121;
  HARDEST := CS323;
  REGIS := [CS221, CS257, CS273];
  REQUIRED := [CS121, CS221, CS257, CS264, CS273]
```

If the members of a set are consecutive elements of the base type, we can use the shorthand

```
REQUIRED := [CS121..CS273]
```

instead of listing all the elements as we did. The empty set is easily specified by not listing any elements, as

```
FAILED := [ ]
```

Notice that sets, like arrays, are built up from simple scalar components; they thus fall into the class of structured data types. Also, as is the case for arrays, the computer needs to know how much space to set aside for storage of the set. (Furthermore, every system has an implementation-dependent limit on how much storage it is willing to allocate to a set-type variable.) For example, suppose you want to declare variables that are sets of integers. You will find that the definition

```
TYPE SETINT = SET OF INTEGER;
```

is *illegal*. If you want to manipulate sets of integers, you must restrict yourself to a subrange; for example,

```
TYPE SUBINT = 1..50;
```

Then you can define

```
TYPE SETINT = SET OF SUBINT;
```

declare

```
VAR A : SETINT;
```

and assign

```
A := [5,10,15]
```

In practice, then, the base types for set-type definitions are almost always the enumerated or subrange types discussed in Chapter 13. (Your computer may or may not allow you to define a type as a SET OF CHAR.)

14-4 SET OPERATIONS

In Pascal, the symbol for the set intersection operation is $*$, for the set union operation is $+$, and for the set difference operation is $-$. Thus, if we have defined the sets (of type SET OF SUBINT)

```
ALL := [1..10];  
ODD := [1,3,5,7,9];  
EVEN := ALL - ODD;  
A := EVEN*ODD;  
B := ODD+EVEN;  
C := B*ALL
```


then we should find that the set EVEN is [2,4,6,8,10], the set A is [], and the sets B and C are identical to ALL. Unfortunately, we cannot use a WRITE(C) command to see its elements directly. (Names of sets are illegal as parameters in READ or WRITE procedures.) But we shall see later how to get around this problem.

Any of the relational operators =, < >, <=, and >= may be used with sets, but the meanings of those operators are somewhat changed to make sense in comparing sets. Thus = is used to indicate set equality (i.e., the sets have the same members) and < > denotes set inequality. For set inclusion, i.e., denoting that one set is a subset of another, the operators <= and >= are used. For example,

[CS221, CS121] >= [CS121]

is TRUE (i.e., is a condition that evaluates to TRUE). Similarly,

[] <= [CS121]

is TRUE; but

[CS121, CS221] >= [CS257]

is FALSE. Notice that the operators < and >, which one might expect to be used for proper subsets, cannot be used with sets in Pascal.

The special operator IN is defined in Pascal to denote set membership. Thus, in the terminology of Section 14-2 we say that CS221 is an object in the set REQUIRED. Correspondingly, in Pascal we write the condition

CS221 IN REQUIRED

and find that it evaluates to TRUE. Sets and the IN operator are useful for expressing complicated conditions in a simpler way. For example, we might write

IF MONTH IN [JAN, FEB, MAR]
THEN (* WEAR A COAT *)

instead of

IF (MONTH = JAN) OR (MONTH = FEB) OR (MONTH = MAR)
THEN (* WEAR A COAT *)

The IN operator can also be used to solve the problem of writing out the elements of a set (at least when the elements of the base type can be written).

For example, to write the members of the set C defined at the beginning of this section:

```
FOR I := 1 TO 50 DO
  (* LOOP THRU THE ELEMENTS OF THE BASETYPE *)
  IF I IN C THEN WRITELN(I);
```

This FOR loop writes just those integers from 1 to 50 that are in the set C.

Here's a complete program that uses sets to pick out those students who will have completed their required courses by the end of the current semester. We shall assume that there are N students and use an array of length N to hold the sets of courses taken by each student. First, we will read in the sets of courses completed by each student, then the sets of courses taken currently. Since we cannot read in the actual course names, we shall assume the data is coded by the location of the character X in the data line, shown in the table below.

X in column	Means course taken
1	CS121
2	CS221
3	CS257
4	CS264
5	CS273
6	CS281

For N = 4, the data might look like:

XX	X	}	Courses previously completed
X	X XX		
XXX	X		
X	XX		
	XX	}	Current courses
X	X		
	X		
X	XX		

This indicates that the first student has completed CS121, CS221, and CS281 and is currently taking CS264 and CS273; the data is read similarly for the other three students. The output of the program will then be

2
4

These are the numbers of the students who will have completed all required courses at the end of the current semester.

EXAMPLE 14-1

```

PROGRAM COURSECHECK( INPUT, OUTPUT );
CONST N = 10;
TYPE COURSES = (CS121, CS221, CS257, CS264,
                 CS273, CS281);
      COURSESET = SET OF COURSES;
VAR STUDENT: ARRAY[1..N] OF COURSESET;
    CH: CHAR; LIST, I: INTEGER; COL: COURSES;
    REQUIRED: COURSESET;
BEGIN
    REQUIRED := [CS121, CS221, CS257, CS264,
               CS273, CS281];
    FOR I := 1 TO N DO (* INITIALIZE ARRAY *)
        STUDENT[I] := [ ];
    FOR LIST:= 1 TO 2 DO (* PROCESS INPUT LISTS *)
        FOR I:= 1 TO N DO
            BEGIN
                FOR COL:= CS121 TO CS281 DO
                    BEGIN
                        READ (CH);
                        IF CH = 'X' THEN
                            STUDENT[I] := STUDENT[I] + [COL]
                        END;
                    READLN
                END;
            FOR I:= 1 TO N DO (* SEE WHO'LL BE DONE *)
                IF (REQUIRED - STUDENT[I]) = [ ] THEN
                    WRITELN(I)
            END.

```

An interesting feature to notice in this program is the use of the values of the enumerated type COURSES as the index values in the FOR loop. By doing this, we can simply add (set union) the current index value to the set of courses STUDENT[I] when an X is read.

Notice also that by declaring an array to have elements that are sets, we have declared a structured type that has components that are themselves of a structured type. In this way, we can build up increasingly complicated data structures. In the next chapter, we shall use this powerful Pascal facility to define very general structures known as *records*. But first we shall see how to use this facility to solve the problem, posed in Section 10-4, of declaring arrays of arrays.

14-5 ARRAYS OF ARRAYS

Recall the problem that we came up against in Chapter 10. To handle character strings we were obliged to declare them to be arrays (or packed arrays). Then, when we wanted to maintain an array of character strings (e.g., an array of names of students in the class), we were stymied by the fact that the compiler would not accept a declaration of the form

```
X:ARRAY ... OF ARRAY ... OF ...
```

but required that the array be OF a defined type.

Now we see the way around this problem. We can define an array that holds a character string to be a new type, just as we defined various sets to be new types. Then we shall have no problem declaring arrays of this new type. For example, here's how we can declare an array of student names.

```
TYPE NAMES = PACKED ARRAY [1..20] OF CHAR;
VAR NAMELIST: ARRAY[1..100] OF NAMES;
```

The following is a reworking of the example of Section 7-8. This was a grade computation problem. Here we have added the extra feature of reading in and storing the students' names, so that the grades printed out may be identified by name as well as identification number.

EXAMPLE 14-2

```
PROGRAM GRADECOMP (INPUT,OUTPUT);
(* PROGRAM TO COMPUTE WEIGHTED TOTAL OF      *)
(* 3 GRADE COMPONENTS FOR EACH OF N          *)
(* STUDENTS (N <= 100).                       *)
(* INPUT DATA: N                             *)
(*      LIST OF 3 WEIGHTS                     *)
(*      3*N GRADE CARDS IN FORMAT:           *)
(* NAME IDENT COMPONENT GRADE                 *)
(* WHERE IDENT IS STUDENT NUMBER (BETWEEN     *)
(* 1 AND N) AND COMPONENT IS 1, 2, OR 3.      *)
TYPE NAMES = PACKED ARRAY[1..20] OF CHAR;
```

```

VAR NAMELIST: ARRAY[1..100] OF NAMES;
    TEMP: NAMES;
    IDENT, COMPONENT, I, J, N : INTEGER;
    GRADE : REAL;
    WEIGHT : ARRAY[1..3] OF REAL;
    TOTGRADE : ARRAY[1..100] OF REAL;
BEGIN
    READ (N);
    FOR I := 1 TO 3 DO
        READ (WEIGHT[I]);
    READLN;
    FOR I := 1 TO 100 DO (* INITIALIZE ARRAYS *)
        BEGIN
            TOTGRADE[I] := 0.0;
            NAMELIST[I] := '
        END;
    FOR I := 1 TO 3*N DO
        BEGIN
            FOR J := 1 TO 20 DO
                READ(TEMP[J]); (* READ NAME INTO
                    TEMP *)
                READLN (IDENT, COMPONENT, GRADE);
                IF NAMELIST[IDENT] = '
                    THEN NAMELIST[IDENT] := TEMP
                    ELSE IF NAMELIST[IDENT] <> TEMP THEN
                        WRITELN ('ERROR IN DATA');
                TOTGRADE[IDENT] := TOTGRADE[IDENT]
                    + WEIGHT[COMPONENT]
                    * GRADE
            END;
        WRITELN(' ID      NAME      GRADE');
        FOR J := 1 TO N DO
            WRITELN(J:5, NAMELIST[J]:22,
                TOTGRADE[J]:10:2)
        END.

```

In the example above, we have done a bit more than simply add an array of names to the student data. We have arranged that each grade entered be identified by the student's name, as well as his or her number. Now we are able to catch possible mistakes in the typed data, by including in the code a check that the name on a grade card matches the name previously stored for that same student number. Notice that the character-at-a-time input of character strings in Pascal has required us to make careful use of READ and READLN so that we get just the character string we want stored in the NAMELIST entries. Also notice that it is most convenient to first read the characters into a temporary location (TEMP) and then move the whole name

as a string from TEMP to the NAMELIST. For efficiency, it would have been better to read the characters into an unpacked array and then pack them (using the procedure PACK) for storage in NAMELIST. (See Exercise 14-1.)

14-6 VARIABLES OF DEFINED TYPES AS SUBPROGRAM PARAMETERS

In Section 11-9, in describing the syntax for subprogram parameter lists, we noted that the identifiers corresponding to the parameters had to be followed by a type *identifier*—i.e., a single word—to specify their type. This precluded our specifying an array as a subprogram parameter at the time, although we certainly want to be able to pass arrays to subprograms for processing. For example, a subprogram might sort an array of names or perform statistical calculations on an array of numbers. We could, of course, declare the array to be global, so that all procedures of the program have access to it. (This was the approach taken in the SORT program developed in Section 12-6.) But a much cleaner approach is to define a new type for the array and use the identifier of this newly defined type in the parameter list. As a simple example of how this works, we have rewritten PROCEDURE MAXMIN (Example 12-3) to include in the parameter list the name and length of the array to be examined.

EXAMPLE 14-3

```
PROCEDURE MAXMIN (LENGTH: INTEGER; A: NUMARRAY;  
    VAR LARGEST, SMALLEST: INTEGER);  
VAR I: INTEGER;  
BEGIN  
    SMALLEST := A[1];  
    LARGEST := A[1];  
    FOR I := 2 TO LENGTH DO  
        BEGIN  
            IF A[I] > LARGEST  
            THEN LARGEST := A[I]  
            ELSE IF A[I] < SMALLEST  
            THEN SMALLEST := A[I]  
        END  
    END;  
END;
```

Now here's how we incorporate the procedure MAXMIN into a complete program. Notice that the type definition, which is needed for both the main program and the subprogram, is global.


```

PROGRAM TMAXMIN ( INPUT, OUTPUT );
CONST N = 10;
TYPE NUMARRAY = ARRAY[1..100] OF INTEGER;
VAR I, L, S: INTEGER; DATA: NUMARRAY;
(* INSERT PROCEDURE MAXMIN HERE *)
BEGIN (* MAIN PROGRAM *)
    FOR I := 1 TO N DO (* LOAD DATA *)
        READLN (DATA[I]);
        MAXMIN(N, DATA, L, S);
        WRITELN(' LARGEST IS' , L);
        WRITELN(' SMALLEST IS' , S)
    END.

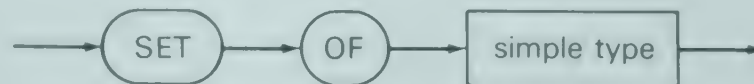
```

Whenever you use an array as a subprogram parameter, you should consider carefully whether it should be a value or a variable parameter. Recall that a copy of a value parameter is made for the use of the subprogram; this protects the original value of the parameter from inadvertent damage. We used this option in Example 14-3 since we wanted to ensure that no changes could be made to the array by the procedure. If the array had been very large, however, copying it would have been costly, both in terms of time and memory space. We would probably then choose to make the array a variable parameter and to be particularly careful in writing the procedure so as not to damage the array.

Although arrays are probably the most common example of a subprogram parameter of a user-defined type, keep in mind that parameters can be of any defined type, including sets or subrange or enumerated types.

14-7 SYNTAX

The only new syntax in this chapter is the addition of type SET to the possibilities for *type*:



where simple type is as defined in Chapter 13. Again, we must remember that the syntax doesn't tell the whole story: there are rather severe limitations on what can be used for "simple type."

14-8 EXERCISES

- 14-1.** Rewrite Example 14-2 so that it reads the input characters into an unpacked array and then packs them for storage.

- 14-2.** Rewrite Example 14-2 so that a procedure is used to input the character string.
- 14-3.** A 3×3 array of characters is used in a simulated tic-tac-toe game. Each array element is initially blank (' ') but may be changed during play to either 'X' or 'O'. Write a procedure that (1) checks whether a given move is legal and, if so, (2) performs that move. Procedure parameters are BOARD (the array), MOVE, ROW, and COL. The value of MOVE is 'X' or 'O' and is to be placed in the array element specified by ROW and COL (which contain integers in the range 1 to 3) only if that element was previously blank. To test your procedure, write a main program that reads descriptions of moves, calls the procedure, and writes the contents of BOARD.
- 14-4.** Let A be an array of 100 REAL numbers. Suppose N numbers have been stored in A[1], A[2], ..., A[N]; ($N < 100$). Write a procedure to insert a value V between the Ith and the (I+1)st elements in this list. [For example, suppose $N = 5$ and the array currently contains 1, 2, 3, 4, 5. After execution of your procedure for $I = 3$, $V = 0.01$, A should contain 1, 2, 3, 0.01, 4, 5. Also notice that N (a variable which the main program uses) should be increased to 6.]
- 14-5.** Write an interactive program that reads a typed line of text and echoes it with the vowels all replaced by asterisks. (Hint: Use a SET with members 'A', 'E', 'I', 'O', and 'U'.)
- 14-6.** Write a program that reads two lists of integers between 1 and 50 and prints out the integers that are common to both lists. (Hint: Create two SETs that contain the integers in the two lists. Do not add an integer to a set if it's already in the set.)
- 14-7.** Write an interactive program to implement the game of "Hangman," in which one player types in a word and the other tries to guess it by guessing one letter at a time. A correct guess should result in a display of the letter in its proper position (or positions) in the word. The person guessing the word can guess N incorrect letters before losing the game. Make N a CONST parameter of the game and adjust it according to the skill of the players. Use a SET to keep track of the letters in the word that haven't yet been guessed.

SUMMARY OF CHAPTER 14

In Pascal we can define a data type, known as SET, whose values are sets of a scalar data type. We can compute the intersection (*), union (+), and difference (-) of sets. We can also compare sets for equality (=), inequality (<>), and inclusion (<= and >=). We can also use the operator IN to test for set membership.

A set is known as a structured type since it is built up from values of simple types. Similarly, an array is a structured type, although its elements need not be single values but can be values of some other structured type (possibly another array). It is therefore convenient to be allowed to define array structures as new, named types.

15

RECORDS AND FILES

15-1 INTRODUCTION

In this chapter we shall examine the most powerful of the structured types, namely, the *record*. The structured types we looked at so far all have one thing in common—they are aggregates of data items that are all of the same type. Thus we had sets of integers or sets of colors, arrays of characters (e.g., NAMES) and arrays of NAMES. But this is not always the most natural way to group data.

Recall that in Example 14-2 we set up an array of names (NAMELIST) and an array of grades (TOTGRADE) to store the names and grades of the students in a class. The name and grade of a particular student were linked through the array index. For example, if NAMELIST[5] contained SMITH and TOTGRADE[5] contained 195, we knew that Smith's grade was 195. Linking related data items in this way is awkward and contrary to the way we'd like to view such data. What we would like to do is to maintain a record, in the normal sense of the word, for each student. This record might contain various items of information about the student, such as name, social security number, curriculum, phone number, grades for several exams and homework, etc. We can imagine such a record as a line in a grade book:

Name	SSN	Curric	Phone	Grade 1	
Joe Smith	582694311	5229	3-2111	52	

In Pascal, we can define structured types, called record types, which are aggregates of different types of data items. For example, we might define a

record type named STUDENT that consists of a 20-character name, followed by an integer social security number, and so on.

Then suppose we have defined a type STUDENT and we want to store information for 50 students. One way would be to declare an array of records of type STUDENT. This is fine for small amounts of data. It becomes impractical, however, for large amounts. Suppose, for example, we need to set up a record for each of the students of a large university. And suppose that each record contains 100 data items. Then altogether the records consist of 3 million data items—far too large an array to store in the memory of a computer. The solution, as we shall see, is to maintain such a collection of records as what is known as a *file*. A file of records is maintained outside main memory; individual records are looked up and transferred into main memory to be processed as needed. The situation is completely analogous to maintaining information on students as paper files, each student's record (a collection of data written on pieces of paper) being stored in a separate manila file folder, with the folders all stored in file cabinets from which they are fetched when needed. Indeed, the terminology used in computer data processing was borrowed from the familiar concepts of records and files as used for many years in manual handling of data.

15-2 DEFINING RECORD TYPES AND DECLARING RECORDS

Defining a type RECORD is a straightforward process. All we have to do is to list, in order, the types of the record components, that is, the data items to be successively stored in the record. Record components may be of any previously defined type, including arrays, subrange types, and even other record types. For example, a record type named STUDENT might be defined as follows:

```
TYPE STUDENT = RECORD
    NAME : NAMES;
    SSN, CURRIC, PHONE : INTEGER;
    GRADE : REAL
END;
```

Here we have assumed that NAMES is a previously defined type; namely, an ARRAY OF CHAR. Any record of type STUDENT will, then, have five components, or *fields*. The first field contains the student's name, the next field the SSN (an INTEGER), and so forth. Notice that we have a single field for GRADE. What we would really like is room in the record for a whole array of grades. Another way we can make our student record description more useful is to use subrange types to limit the allowable entries in the INTEGER fields. For example, we may know that all campus phone numbers are of five digits, beginning with "2" or "3." By more closely specifying the type of a data item, we can catch some of the errors that will inevitably occur when data is typed into the computer.

Here, then, is another definition of a student record, complete with the necessary definitions of types used in that record.

EXAMPLE 15-1

```

TYPE NAMES = PACKED ARRAY[1..20] OF CHAR;
   GRADES = ARRAY[1..20] OF REAL;
   DATE = RECORD
       MO : 1..12;
       DAY : 1..31;
       YEAR : 1..99
   END;
STUDENT = RECORD
    NAME : NAMES;
    SSN : INTEGER;
    ADMITDATE : DATE;
    PHONE : 20000..39999;
    CURRIC : 1000..9999;
    CLASS : (FRESH, SOPH, JR, SR);
    GRADELIST : GRADES
END;
```

Notice that the subrange types do not have to be separately defined but can be defined within the record definition. Thus we have defined PHONE to be of a type that is a subrange of the integers. Similarly, CLASS is an enumerated type, with values to be drawn from the list (FRESH, SOPH, JR, SR). If we do not want to use those types elsewhere in our program, we can also nest the definitions of NAMES, GRADES, and DATE within our student record, as follows:

```

TYPE STUDENT = RECORD
    NAME : PACKED ARRAY[1..20] OF CHAR;
    SSN : INTEGER;
    ADMITDATE : RECORD
        MO : 1..12;
        DAY : 1..31;
        YEAR : 1..99
    END;
    PHONE : 20000..39999;
    CURRIC : 1000..9999;
    CLASS : (FRESH, SOPH, JR, SR);
    GRADELIST : ARRAY[1..20] OF REAL
END;
```

Clearly, we can build up record types of any degree of complexity in this manner. However, there is one drawback that we can see here. All student

records must look exactly alike; that is, they must contain exactly the same fields. We all know that all students are different and some will have special problems. We need a way to include additional, different data items in certain individual records. (It would be wasteful to allow extra room for these items in all the records.) In Pascal, this flexibility is provided by the inclusion of a so-called *variant part* in a record. The variant part is made up of *variants*, each of which consists of a number of fields. For each particular instance of the record, one of the variants is selected. One of the fields of the record, called the *tag field*, is used to select the variant. The different variants are then described in a *CASE clause* resembling, but not to be confused with, the *CASE statement*. The different cases, describing different variants, are associated in the usual way with different values of the CASE variable—which in this situation is the tag field.

As an example, consider the problem of computing a payroll. Employees are of two types—salaried workers who are paid a fixed amount monthly and hourly workers who are paid according to how many hours they have worked. For the salaried workers, all we need to store to generate a paycheck is the salary. For the “hourlies,” we must store base pay rate, overtime rate, and hours worked. Assuming that we have first defined

```
TYPE WORKERTYPE = (SALARIED, HOURLY)
```

an employee record type might, then, look something like:

```
TYPE EMPLOYEE = RECORD
  NAME : PACKED ARRAY [1..20] OF CHAR;
  SSN  : INTEGER;
  CASE W : WORKERTYPE OF
    SALARIED : (SALARY : REAL);
    HOURLY   : (BASERATE, OVRATE, HOURS : REAL)
  END;
```

In this example, W, a variable of type WORKERTYPE, is the tag field. When employee data is read into a record of this type, the value of W determines whether one field (SALARY) or three fields (BASERATE, OVRATE, HOURS) are included in the variant part. A field is, of course, also set up to hold the value of W.

There are two important rules to remember about defining records with variant parts:

1. The list of fields in a record (i.e., the *field list*) can have only one variant part.
2. The definition of the variant part must appear *after* all of the other fields; i.e., *after* the fixed part.

We end this section with two more examples.

```

TYPE PERSON = RECORD
    NAME : NAMES;
    BIRTHDATE : DATE;
    CASE MARRIED : BOOLEAN OF
        TRUE : (SPOUSE : NAMES);
        FALSE : ( );
    END;

```

Here we include a field for the spouse's name only when the tag field MARRIED takes on the value TRUE. Notice that the empty field list for the case FALSE is denoted by ().

```

TYPE PERSON = RECORD
    NAME : NAMES;
    BIRTHDATE : DATE;
    CASE MARRIED : BOOLEAN OF
        FALSE : ( );
        TRUE : (SPOUSE : NAMES;
                CASE INSURED : BOOLEAN OF
                    FALSE : ( );
                    TRUE : (AGE : 0..99;
                           DEPENDENT : BOOLEAN));
    END;

```

Here, in addition to having a variant part that depends on whether the person is married, a variant within this variant includes extra fields when the spouse is insured. This does not violate the rule that a field list can have only one variant part. What we are dealing with here is simply nesting. The field list at what we might call the record level does indeed include just one variant part. But the field lists describing the different variants may then themselves each contain one variant part.

Once we have described a record type in a type definition, we can declare instances of that type in the usual way. For example, to declare records of type STUDENT for two students, we can write

```

VAR JOE, MARY : STUDENT;

```

JOE and MARY are now the names of two variables of type STUDENT.

We can also declare arrays of records:

```

VAR CLASS : ARRAY [1..100] OF STUDENT;

```

Now we have 100 records of type STUDENT and we have named them CLASS[1], CLASS[2], . . . Since the names of records appear explicitly in declara-

tions, there seems to be no problem in referencing a record. Indeed, we can write assignment statements like

```
JOE := MARY
```

meaning that each field of the variable JOE is to be assigned the value of the corresponding field of variable MARY. But just as is the case for other structured data types, we cannot READ or WRITE a record in its entirety. For example, we can't type all of the data on Joe on a line and input it with a simple

```
READLN( JOE )
```

Just as we must read data into arrays one component at a time, we must read data into records one field at a time. This means we must have some way of talking about—or referencing—the individual fields of a record, such as the SSN field of record JOE (i.e., Joe's SSN). In the next section, we discuss how this problem is solved in Pascal.

15-3 REFERENCING RECORDS AND THEIR FIELDS

As noted in the last section, all records to be used in a program are declared in the same way as variables and hence have a name—JOE, MARY, CLASS[10], or whatever. But it is the individual data items, or fields, that are of interest. If we have an array CLASS of student records, we may want to WRITE the grades of a particular student, look up the SSN of a student whose name is JOE SMITH, determine the average grade for the class in the final exam, count how many students got A's, etc. In each one of these examples, we must examine certain fields of the records in the array and not the records themselves as entities.

To reference a component of a record in Pascal, the record identifier is followed by a point, or period, which is then followed by the field identifier. For example, to assign a value to the field SSN in record JOE, we write

```
JOE.SSN := 588921034
```

Similarly, we might write

```
CLASS[5].PHONE := 36684
```

to assign a phone number to the appropriate field in the record of the fifth member of the class. Similarly, we can reference the fields in other contexts, such as reading or writing:

```
READLN(CLASS[10].CURRIC);  
WRITELN(CLASS[10].CURRIC)
```

or computing the average final exam grade for the students in the class:

```
TOTAL := 0;  
FOR I := 1 TO 100 DO  
    TOTAL := TOTAL + CLASS[I].FINAL;  
AVERAGE := TOTAL/100
```

In short, we can use a field of a record in any situation where referencing the simple data type contained in that field is legal.

If records are nested within records, it requires multiple usage of the point notation to work down to the field (or simple data item) level. Thus, to store the admission date of the tenth student in the class we write

```
CLASS[10].ADMITDATE.MO := 8;  
CLASS[10].ADMITDATE.DAY := 20;  
CLASS[10].ADMITDATE.YEAR := 81
```

And if an array is nested inside a record, we must, as usual, specify an index value:

```
CLASS[10].GRADELIST[5] := 95.6
```

When we want to make a number of references to the same record (for example, to read or assign values to all of its fields), writing the record identifier over and over gets tedious. Fortunately, Pascal provides us a shortcut, namely, the WITH statement. By using the WITH statement we can specify the record identifiers to be understood within the statement. Thus we might write:

```
WITH CLASS[10] DO  
    BEGIN  
        SSN := 588216540;  
        PHONE := 20155;  
        CURRIC := 1525  
    END;
```

This assigns values to the specified fields of the record CLASS[10]. Since the WITH statement is just another statement, it can be nested within another WITH. This facility is useful when records are nested within records. Here is the above example expanded to assign values to the fields within the sub-record ADMITDATE.

```
WITH CLASS[10] DO  
    BEGIN  
        SSN := 588216540;  
        PHONE := 20155;  
        CURRIC := 1525;  
        WITH ADMITDATE DO
```



```
BEGIN
    MO := 8;
    DAY := 20;
    YEAR := 81
END
END;
```

As a form of shorthand, we can simply list in order the record names that are to be assumed. Thus we can shorten the above example to

```
WITH CLASS[10], ADMITDATE DO
    BEGIN
        SSN := 588216540;
        PHONE := 20155;
        CURRIC := 1525;
        MO := 8;
        DAY := 20;
        YEAR := 81
    END;
```

Names of fields within a record must be unique; if they were not we wouldn't know which of the fields with the same name we were talking about. This rule seems to pose a problem if, for example, we want to include two dates, ADMITDATE and BIRTHDATE, within student records. It appears that then we would have two fields named DAY, two named MO, and two named YEAR. However, the pairs are readily distinguishable since they are components of two separate records, ADMITDATE and BIRTHDATE, nested within the student record. The two different fields named DAY are referenced as ADMITDATE.DAY and BIRTHDATE.DAY or, using the WITH, when we write

```
WITH ADMITDATE DO
    DAY := 14
```

DAY then refers to the component of ADMITDATE, while when we write

```
WITH BIRTHDATE DO
    DAY := 25
```

DAY refers to the component of BIRTHDATE.

The rule in general is that any naming scheme is legal as long as it is clear what we are talking about. Thus it is even legal to write

```
TYPE NEW : RECORD
    A : INTEGER;
    B : INTEGER
END;
VAR A : NEW;
```

since the record A and the integer field A.A in the record A cannot be confused. Indeed, we have done something very like this when we included a field CLASS in our STUDENT record and then declared an array CLASS of these records.

To illustrate the points we have been making so far in this chapter, here's a complete program that sets up an array of student records and reads data into them. We shall assume that the input data for each student begins with an integer identifier that specifies the location (index) in the array for that student's data. After reading the data in, the program scans the array for students that are seniors and prints their names. Since it should not be necessary to completely fill the array, the SSN field of each record is initialized to zero for easy recognition of empty records.

EXAMPLE 15-2

```

PROGRAM EX152 (INPUT, OUTPUT);
TYPE NAMES = PACKED ARRAY[1..20] OF CHAR;
    DATE = RECORD
        MO: 1..12;
        DAY: 1..31;
        YEAR: 1..99
    END;
    STUDENT = RECORD
        NAME: NAMES;
        SSN: INTEGER;
        ADMITDATE: DATE;
        PHONE: 20000..39999;
        CURRIC: 1000..9999;
        CLASS: (FRESH, SOPH, JR, SR);
        BIRTHDATE: DATE
    END;
VAR I: INTEGER;
VAR CLASSCODE: 1..4;
VAR CLASS: ARRAY[1..100] OF STUDENT;
PROCEDURE READNAME (VAR STRING: NAMES);
    VAR I: INTEGER;
        CHARS: ARRAY[1..20] OF CHAR;
    BEGIN
        FOR I := 1 TO 20 DO
            READ(CHARS[I]);
        READLN;
        PACK(CHARS, 1, STRING)
    END;
BEGIN
    FOR I := 1 TO 100 DO
        CLASS[I].SSN := 0;

```

```
READLN(I);
WHILE I <> -1 DO BEGIN
    WITH CLASS[I] DO BEGIN
        READNAME(NAME);
        READLN(SSN);
        WITH ADMITDATE DO
            READLN(MO, DAY, YEAR);
        WITH BIRTHDATE DO
            READLN(MO, DAY, YEAR);
        READLN(PHONE, CURRIC, CLASSCODE);
        CASE CLASSCODE OF
            1: CLASS := FRESH;
            2: CLASS := SOPH;
            3: CLASS := JR;
            4: CLASS := SR
        END
    END;
    READLN(I)
END;
FOR I := 1 TO 100 DO
    IF CLASS[I].SSN <> 0 THEN
        IF CLASS[I].CLASS = SR
            THEN WRITELN(CLASS[I].NAME)
END.
```

15-4 FILES

In the preceding sections, we described how to set up and use arrays of records to hold all of the records for, say, the students in a class or the employees of a company. There are, however, serious limitations to this approach. First is the matter of memory space. Arrays are always stored in their entirety in main memory. Although a large computer may have a main memory with millions of locations (and the memory of even a small personal computer may have tens of thousands of locations), real-life applications can quickly exceed the capacity of even the largest computer memory. Consider, for example, the massive amount of information maintained by a large bank, or by a large corporation with 10,000 employees, or by the U.S. Internal Revenue Service, or even by a large university that must keep track of all the grades and other data on tens of thousands of students.

Moreover, even if our computer memory is large enough to store all our data in a gigantic array, there are good reasons why we don't want to do so:

1. The number of records changes in time, but we must declare an array of fixed size to hold the records. So either we declare the array to be larger than we'll ever need (thus wastefully leaving much empty space) or we are in danger of running out of space and having to rewrite our pro-

grams with larger array declarations. Clearly, a more flexible arrangement would be preferable.

2. If we turn the computer off (or there is a power outage), the contents of the memory disappear. Hence all computer systems have auxiliary memories, alternative storage devices that can hold data and programs permanently—and also relatively cheaply.

For these reasons, it only makes sense to maintain any large number of records in auxiliary memory, reading them into the computer's main memory as needed.

Let us look more closely at these auxiliary storage devices. The ones in most common use store data on a magnetic surface, just like music is recorded on a cassette tape. Indeed, one of the earliest computer auxiliary storage media developed was magnetic tape that is much like your cassette tape, but, being wider and thousands of feet long, is stored on a large reel. Although tape storage, being very cheap, is still very much used, it has the disadvantage that it is not easy to locate a particular data item on a tape. (Try stopping on a specific note in the middle of one of your cassette tapes!) For this reason, most systems make heavy use of magnetic disks that look much like your stereo records—except that the data is stored on concentric magnetic tracks on the disk surface. By recording the data on separate tracks, instead of one long spiral, it is easy to locate a particular block of data. Magnetic tapes—and most disks as well—also have the advantage of portability: that is, you can store your data or program on a tape or disk and carry it with you for use on a different computer system. For example, Mars data from a space probe might be picked up, processed, and written to a tape by a computer in Houston. The tape might then be mailed to California to be used by a sophisticated computer system that draws a map of Mars in color.

A file (most notably a file of records) is a group of items to be stored in auxiliary memory. Although files can have a more complicated structure, the files we will be talking about in this book are just simple sequences of components. (Hence they are called *sequential files*.) Such a file looks much like an array of records, but without an index. Therefore you cannot find a particular record directly, but must look at the records one at a time, starting at the top. This is, of course, the most sensible sort of organization for a file on magnetic tape, since data on magnetic tapes is naturally read in sequence as the tape passes through the read-write head. And even though skipping around is much more feasible on a disk, it is again a sequence on a track that is read as the disk rotates.

The idea of a file is probably not new to you. If you have been using a terminal to enter your program into the computer, you have found that you had to type your program as a file. Similarly, you may have created data files made up of the successive lines of data to be input. In either one of these cases the file looks like one long character string, to be processed by the program reading it. In this section, we shall explain in general how to declare and use files, whether they have components that are simple types like `INTEGER` or `REAL` or whether they are made up of records with complex structures, such

as we have been discussing in this chapter. (Files of characters, however, have some special aspects; we shall consider them in the next section.)

To declare a file named CLASS of records of type STUDENT, we write:

```
VAR CLASS: FILE OF STUDENT;
```

Similarly, for a file of records of type EMPLOYEE:

```
VAR EMPFILE: FILE OF EMPLOYEE;
```

Files can have components of any standard or defined type. For example:

```
VAR INTLIST: FILE OF INTEGER;
```

In this case, each component of the file is a single integer value.

Now to write to or read from a file, there must be some way to keep track of where we are in the file. (Remember that file components are not numbered by an index value as array components are!) For this purpose the system maintains a *buffer variable*, a variable whose value at any time is that of one of the components of the file. The buffer variable is often described as a *window* through which a component of the file may be seen. By moving the window, we can see the successive components of the file in turn. If the file is named F, for example, then the associated window, or buffer variable, is denoted $F \uparrow$.

Since we generally do not know how many components are in a file, we need a way to tell when we have come to the end of the file. We can imagine that at the end of a file, following all the components of the file, there is a marker called the *end-of-file marker*. Pascal provides a boolean function EOF that returns the value TRUE when the window has been moved past all the components of the file and is over the end-of-file marker, and returns the value FALSE otherwise. EOF has as its single parameter the name of the file being examined. Thus, for example, let CLASS be a file. Then EOF(CLASS) is TRUE if the window is over the end-of-file marker of the file CLASS. On the other hand, EOF(CLASS) is FALSE if the window is over any component (*not* the end-of-file marker) of the file CLASS.

Now let us look at the commands provided in Pascal to move the window and to read from and write to the file. First, suppose that we have declared a file F to be a FILE OF INTEGER and we want to enter data into it. The command to use is PUT(F). Since the file is initially empty, EOF(F) is TRUE and $F \uparrow$ is undefined (since no record is visible through the window). The initial situation can be pictured as in Figure 15-1.

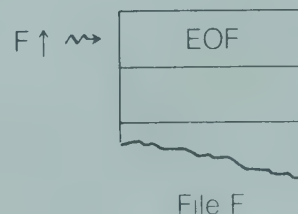


Figure 15-1

Note that the file is empty except for the end-of-file marker EOF at the top of the file. Since $F \uparrow$ is referencing this spot, its value is undefined. By assigning a value to $F \uparrow$ and executing `PUT(F)`, we cause that value (i.e., the value of $F \uparrow$) to be written into the first position in the file. `PUT(F)` also moves the window ahead one spot, so `EOF(F)` remains TRUE and we are ready to write to the next position in the file. Thus the commands

```
F↑ := 52;
PUT(F)
```

will change the situation to that shown in Figure 15-2.

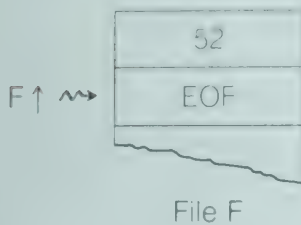


Figure 15-2

As a further example of how this works, here's a FOR loop that writes the integers 1 through 5 to the file F.

```
FOR I := 1 TO 5 DO
  BEGIN
    F↑ := I;
    PUT(F)
  END
```

After this loop executes, `EOF(F)` will still be TRUE and the window undefined, so we are ready to append more data to the file by further `PUT` commands. We can picture the file as shown in Figure 15-3.

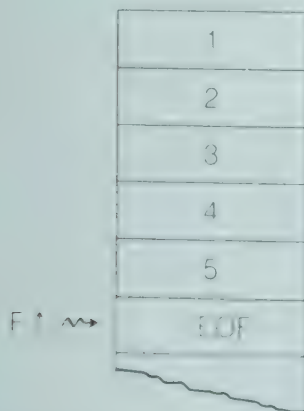


Figure 15-3

Here we show $F \uparrow$ now referencing the sixth position in the file, which contains the end-of-file indicator. As we have seen, as we write to a file the end-of file marker is automatically moved along to delimit our data.

Now suppose that we want to print the contents of this file so that we can see that it really does contain the integers 1 through 5. First, we must move the window back up to the top of the file. We do this with the command

```
RESET(F)
```

After execution of this command, the situation looks like that shown in Figure 15-4.

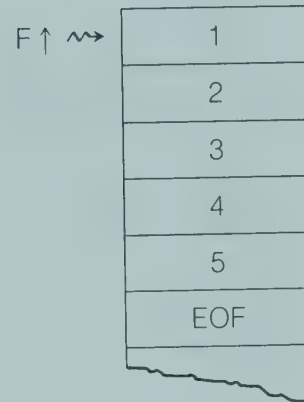


Figure 15-4

Now we can “see” the first file component through the window; specifically, the value 1 is assigned to the buffer variable $F \uparrow$. So we know how to print out the first file entry—just use

```
WRITELN(F↑)
```

Furthermore, $EOF(F)$ is now FALSE, since what we see through the window is not the end-of-file marker. Then to move down through the file, we use the command

```
GET(F)
```

which moves the window to the next entry [i.e., assigns the next value (2) to the buffer variable $F \uparrow$]. If a GET command moves the window past the end of the file, $EOF(F)$ again becomes TRUE. Here then is a program segment that prints out our file of integers.

```
RESET(F);
FOR I := 1 TO 5 DO
  BEGIN
    WRITELN(F↑);
    GET(F)
  END;
```

After this code executes, the window will again be at the end-of-file marker. Notice that the loop index here plays no role except to keep count of the

number of items as they are printed. And in general we are not supposed to know how long a file is. We should really have used a WHILE loop that takes advantage of the EOF condition as follows.

```

RESET(F);
WHILE NOT EOF(F) DO
  BEGIN
    WRITELN(F↑);
    GET(F)
  END;

```

To put these ideas together, here's a program that reads an integer N, loads a file with the integers 1 through N, and then prints out the contents of the file.

EXAMPLE 15-3

```

PROGRAM EXFILE (INPUT, OUTPUT);
VAR I, N: INTEGER;
    F: FILE OF INTEGER;
BEGIN
  READLN(N);
  FOR I := 1 TO N DO
    BEGIN
      F↑ := I;
      PUT(F)
    END;
  RESET(F);
  WHILE NOT EOF(F) DO
    BEGIN
      WRITELN(F↑);
      GET(F)
    END
  END.

```

There is one more command that you should know about:

REWRITE(F)

Use this when you want to start over and fill your file with new data. When REWRITE(F) is executed, the file F is emptied, EOF(F) becomes TRUE and the window is positioned at the top of the file. You are then ready to write new data to the file. The situation can be pictured as in Figure 15-5, which is just what the file looked like when it was originally declared.

As a variation on Example 15-3, here's an interactive program that accepts a list of integers (terminated by 0) from the user, stores them in a file, and computes the average of the integers for the user. The REWRITE com-



Figure 15-5

mand is then used to clear out the file, so the program is ready to accept a new list. (Of course, there is no need to store a list of numbers in a file if all you want to do is to compute their average; imagine some more complicated computation.)

EXAMPLE 15-4

```

PROGRAM AVINT (INPUT/,OUTPUT);
VAR COUNT, TOTAL, N: INTEGER;
    F: FILE OF INTEGER;
    ANSWER: CHAR;
BEGIN
    ANSWER := 'Y';
    WHILE ANSWER = 'Y' DO
        BEGIN (* INTERACTIVE LOOP *)
            WRITELN ('ENTER LIST OF INTEGERS, THEN 0');
            READLN;
            READ(N);
            WHILE N<>0 DO
                BEGIN (* READ LIST *)
                    F↑ := N;
                    PUT(F);
                    READ(N)
                END;
            RESET(F);
            COUNT := 0;
            TOTAL := 0;
            WHILE NOT EOF(F) DO
                BEGIN (* COMPUTE AVERAGE *)
                    COUNT := COUNT + 1;
                    TOTAL := TOTAL + F↑;
                    GET(F)
                END;
            WRITELN('AVERAGE IS ', TOTAL/COUNT:12:3);
            WRITELN;
            WRITELN('DO YOU WANT TO ENTER A NEW LIST?');
            WRITELN ('ENTER Y FOR YES OR N FOR NO');
            READLN;

```



```
      READ (ANSWER) ;  
      REWRITE (F)  
      END  ( * INTERACTIVE LOOP * )  
END .
```

In this example you can see one of the advantages of using a file rather than an array. There was no need to set a limit on how many numbers the user could enter, as we would have had to do if we had used an array. The user can enter 10 integers, or 100, or 1000, or any number, limited only by the amount of file space that the system will allow.

We have said that another advantage of using files is that, being stored on disk, they do not disappear when the program completes execution, as arrays in main memory do. However, this is not true of the file *F*. We have declared it within the program *AVINT*; we have made no provision for it to be saved by the system. The file *F* is therefore an *internal* file, *local* to *AVINT* (in the same sense that any variable is local to a procedure or a program), and will indeed vanish when *AVINT* completes execution.

How, then, do we create *external* files, i.e., files that are saved by the system for later use in some other program? The answer to this question is system-dependent. We will therefore confine ourselves to discussing what has to be done, so that you will know what to look for. First, note that, just as internal files are handled much like local variables, external files are treated similarly to formal parameters within a procedure. Thus one would expect to find the external files used by a program listed in the program header. External files are not unfamiliar to you. When we have discussed program headers, we have merely said that the word *INPUT* must be included in the header if the program reads data, and the word *OUTPUT* must be included if the program sends data to some output medium. But in fact, *INPUT* and *OUTPUT* are two external files that are managed by the system. (Consequently, they do not have to be declared as files by the programmer.)

To show you an example of a user-created external file, we shall outline a program that accepts data on students from a terminal and stores this data in a file *CLASS* of records of type *STUDENT*. It might begin:

```
PROGRAM READIN ( INPUT , CLASS ) ;  
TYPE STUDENT = RECORD
```

```
 .  
 .
```

```
VAR CLASS: FILE OF STUDENT ;
```

Then a companion program that gets data from the file *CLASS* and prints a report on student performance might begin:

```
PROGRAM REPORT ( CLASS , OUTPUT ) ;  
TYPE STUDENT = RECORD
```

```
VAR CLASS: FILE OF STUDENT;
```

The files INPUT and OUTPUT can be handled by the system in a standard (default) manner. But in the case of the file CLASS, you must provide the system with the actual parameter; i.e., the name by which the system knows the file. How you do this depends on your local system. If you have previously specified file names for INPUT and OUTPUT, you should find that you specify a system file name for the file CLASS in essentially the same way.

In Section 15-7 we shall provide you with a rather realistic example of file handling; namely, a set of programs that could be used to create and maintain a file of student records—but first let us look briefly at a special type of file known as a *textfile*.

15-5 TEXTFILES

A *textfile* is simply a file whose components are characters. Thus we might define:

```
TYPE TEXT = FILE OF CHAR;
```

But on systems following the Pascal standard, the type TEXT is predefined (like INTEGER or REAL), so that to declare a textfile we need only write

```
VAR BOOK: TEXT;
```

and a file named BOOK, in which we can store characters, is set up.

Besides all the printable characters, a textfile usually contains two special nonprintable characters—the *end-of-line character*, denoted EOL, and the *end-of-file character*, denoted EOF. Textfiles are divided into lines by the end-of-line character and are terminated by the end-of-file character. Thus a textfile can be pictured as one long string of characters, including occasional EOL characters and terminating, like any file, with the EOF character.

The standard files INPUT and OUTPUT are textfiles, so you are really quite familiar with this type of file. In particular, although the EOL character is not itself printable, you have certainly seen its effects. An EOL in an OUTPUT textfile sent to a printer or to a terminal causes the subsequent character to be printed (or displayed) at the beginning of the next line. And the way you insert EOL characters where you want them in your output is to use the procedure WRITELN; the LN actually indicates that an EOL is to be inserted in the file following the items to be written. Similarly, you insert an EOL in an INPUT file by hitting the “return” key on your terminal keyboard. And to read a whole line (i.e., all the characters up through the next EOL), you use the procedure READLN.

The procedures READ, READLN, WRITE, and WRITELN have been supplied to make your task of reading and writing textfiles easier. Since you may want to process textfiles other than the standard ones INPUT and OUTPUT, it is possible to include a file name as one of the parameters of these procedures. For example,

```
WRITE (OUTFILE, ITEM1, ITEM2)
```

will cause the values of ITEM1 and ITEM2 to be written to OUTFILE. If no file name is given, the file OUTPUT is assumed. Similarly, a file may be specified in a call of the procedure READ:

```
READ (INFILE, A, B)
```

But INPUT is assumed if no file is specified.

It is, of course, possible to use GET and PUT with textfiles just as we described in the previous section. Indeed, the procedures READ, READLN, WRITE, and WRITELN make use of these more primitive operations. What is visible through the window when a GET executes is just one character. Thus the procedure READ reads numbers by using GET to look at characters one by one, skipping blanks and EOLs, recognizing INTEGER values as strings of digits (and, similarly for REAL values), and matching the items read with its parameter list. Since numbers are not represented inside the computer as strings of characters, a conversion must also take place. To give you some understanding of what happens, here is a procedure that scans a textfile T for an integer and converts the characters that represent the integer into an INTEGER value. As an example of how this works, suppose the first character read is '9'. The character '9' is then converted to the number 9 (by means of the ORD function) and saved. If the next character read is a blank, then 9 is returned as the integer. If instead a '7' is read, 9 is multiplied by 10 and '7' is converted to 7 and added, giving the INTEGER 97. If the digits are terminated by a decimal point instead of some other character, this indicates that somebody entered a REAL instead of an INTEGER (type error!).

EXAMPLE 15-5

```
PROCEDURE READINT (VAR T: TEXT; VAR I: INTEGER);
VAR CH: CHAR;
BEGIN
    I := 0;  (* INITIALIZE INTEGER TO 0 *)
    RESET(T);
    CH := T↑;
    WHILE (CH < '0') OR (CH > '9') DO
        BEGIN
            GET (T);
```



```

    CH := T↑
  END;
  WHILE (CH >='0') AND (CH <= '9') DO
    BEGIN (* READ DIGITS *)
      I := I*10 + (ORD(CH) - ORD'0');
      GET (T);
      CH := T↑
    END;
  IF CH='.' THEN
    WRITELN (' YOU GAVE ME A REAL! ')
  END;
END;

```

To test this procedure, we inserted it into the following program that reads 50 characters into the file T, calls READINT to find the first string of digits, converts them to an integer, and then prints the integer.

```

PROGRAM TEST (INPUT, OUTPUT);
VAR T:TEXT;
    I,N: INTEGER;
    (* INSERT READINT HERE *)
BEGIN
  FOR I := 1 TO 50 DO
    BEGIN
      READ (T↑);
      PUT(T)
    END;
    READINT(T,N);
    WRITELN(N)
  END.

```

There are three points to notice about this example. First, we have used the buffer variable $T↑$ as a parameter in a READ; this is another situation in which a buffer variable may be used just like any other variable. Second, we have specified the file T as a VAR parameter in the procedure, even though the procedure is not to make any changes in the file. Since it just doesn't make sense for a local copy of a whole file to be made for the benefit of a procedure, files are not allowed to be value parameters. Third, READINT ignores the EOL marker just as it does any nondigit. This procedure skips from line to line looking for an integer, very much like the procedure READ would. In Example 15-6 below, we have modified READINT so that it looks for an integer only on the current line of text. To stop reading at the end of a line we use the special condition EOLN, which tests for whether the EOL is in the window in exactly the same way the EOF tests for the end-of-file marker. To make this procedure behave analogously to READLN, we have also arranged that, after reading an integer, it reads (and discards) the rest of the characters on that line.

EXAMPLE 15-6

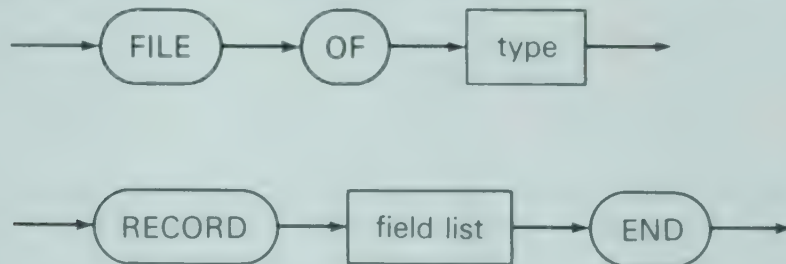
```

PROCEDURE READINT2 (VAR T: TEXT; VAR I: INTEGER);
VAR CH: CHAR;
BEGIN
    I := 0;
    RESET(T);
    CH := T↑;
    WHILE ((CH < '0') OR (CH > '9')) AND (NOT
        EOLN(T)) DO BEGIN
        GET(T);
        CH := T↑;
    END;
    IF EOLN(T) THEN WRITELN ('NO INTEGER FOUND')
    ELSE BEGIN
        WHILE (CH >= '0') AND (CH <= '9') DO BEGIN
            I := I*10 + (ORD(CH) - ORD('0'));
            GET(T);
            CH := T↑;
        END;
        IF CH = '.' THEN
            WRITELN ('YOU GAVE ME A REAL!')
        ELSE (* READ REST OF LINE *)
            WHILE NOT EOLN(T) DO GET(T);
    END
END.

```

15-6 SYNTAX

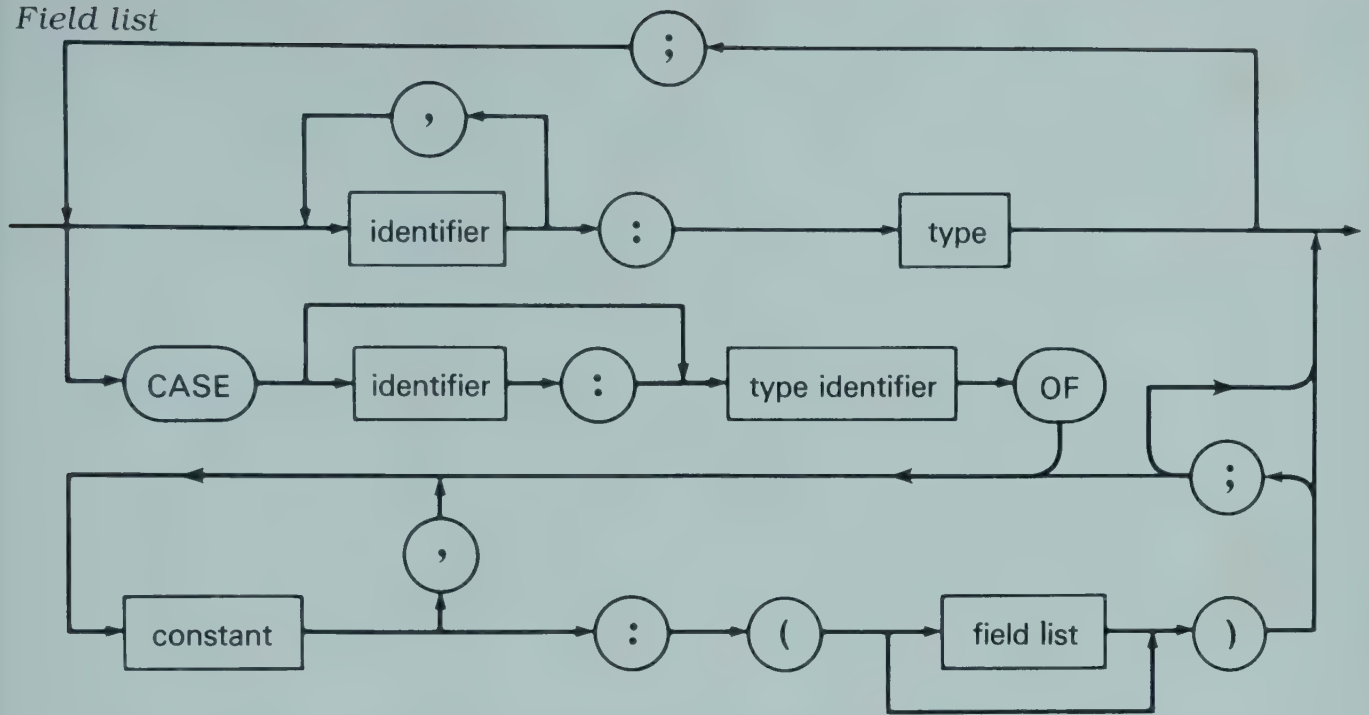
In this chapter we have made several additions to your knowledge of Pascal syntax. First, we have added two new possibilities for *type*:



As you can see by consulting the syntax diagram for *type* in Appendix B this completes the possibilities for *type*. Notice that the syntax allows for defining files whose components are files. Your system is unlikely to allow this, however. Setting up a buffer variable to hold a file component that is a whole file is just not very feasible!

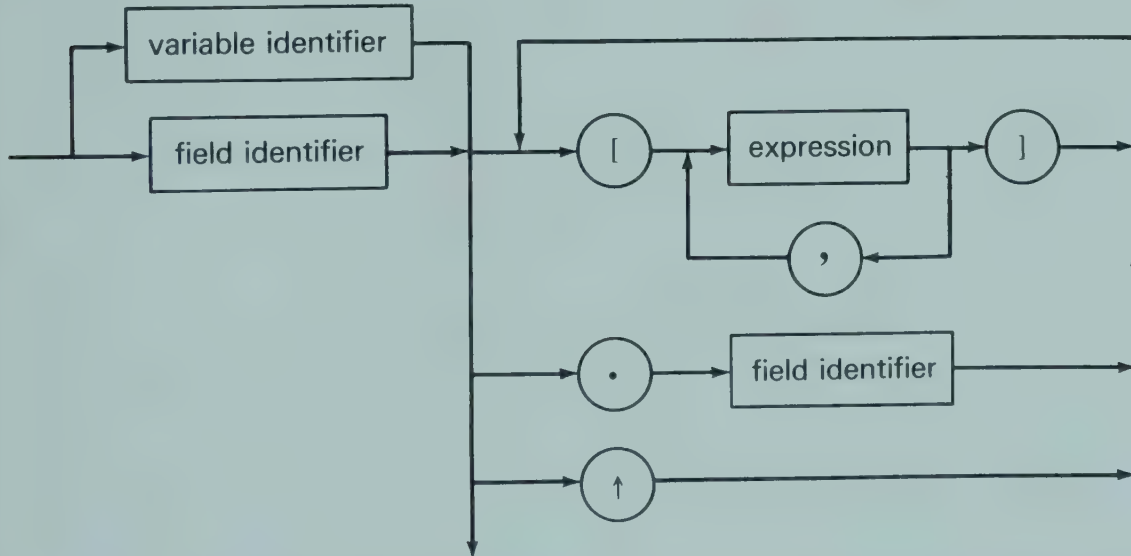
To complete the syntax for the type RECORD, we need to specify the syntax for *field list*.

Field list

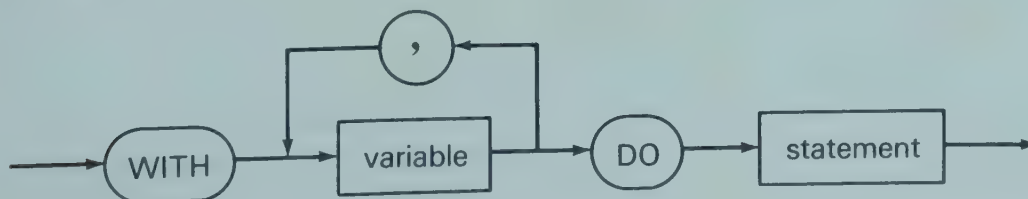


Next we must expand the possibilities for variable to include the referencing of record fields and of buffer variables. The complete syntax for *variable* then becomes:

Variable



Also, there is the alternative way to reference fields of a record, namely, the *WITH statement*:



In working with records and files, you must be particularly careful to keep the *semantics* as well as the *syntax* in mind. Semantically correct constructs for *variable* will only make sense in the appropriate contexts. For example, the construct



is not meaningful for an arbitrary identifier, but is meaningful for a "variable identifier" that is the name of a file. Similarly, the WITH statement will make sense when the "variable" is the name of a record.

The file operations GET, PUT, RESET, and REWRITE are, like READ and WRITE, procedures, and hence they are not included within the Pascal language syntax. Similarly, the special conditions EOF and EOLN are in reality standard functions that return BOOLEAN values.

15-7 A PROGRAMMING EXAMPLE: STUDENT RECORDS

At the beginning of each semester we want to set up a file CLASS of student records in which to accumulate the grades of all of the students as the term progresses. Our first problem is to determine exactly what data is to be placed in a student's record. This depends upon how many exams we plan to give, how many homework assignments we plan to make, etc. We also need to decide what personal information (name, SSN, class, etc.) we want to maintain for each student. After careful consideration of our needs, we decided to define our student record as follows.

```

TYPE STUDENT = RECORD
  SSN: INTEGER;
  NAME: PACKED ARRAY[1..20] OF CHAR;
  CURRIC: 1000..9999;
  EXAM1: 0..100;
  EXAM2: 0..100;
  HOMEWORK: ARRAY[1..10] OF REAL;
  GRADE: CHAR
END;
```

The first task that we need to accomplish with this file is to load it with the students' personal data at the beginning of the semester. There are various ways we could do this. But perhaps the simplest way is to type the student data line by line, one line for each student, in an INPUT file, and then write a program that reads data from INPUT and stores it appropriately in CLASS. Looking at what we need to enter, we decide to format each line of INPUT as follows.

The classcode will be an integer (1, 2, 3, or 4) that will be converted to FR, SO, JR, or SR for storage in the record. Program INITLOAD, given below, then performs the task of reading data in this format and storing it into the STUDENT records. (Note that for simplicity we have omitted the definition of type STUDENT and only indicated where it is to be inserted in the program.) We want the file CLASS to be external, so in executing this program we are careful to give CLASS a system name and to tell the system to save it.

```
PROGRAM INITLOAD (INPUT, CLASS);
(* INSERT STUDENT TYPE DEFINITION HERE *)
VAR CLASS: FILE OF STUDENT;
    CLASSCODE: 1..4;
    C: ARRAY[1..20] OF CHAR;
    I: INTEGER;
BEGIN
    WHILE NOT EOF(INPUT) DO
        BEGIN
            READ(CLASS↑.SSN);
            FOR I := 1 TO 20 DO
                READ C[I];
            PACK (C, 1, CLASS↑.NAME);
            READLN (CLASS↑.CURRIC, CLASSCODE);
            CASE CLASSCODE OF
                1: CLASS↑.CLASS := FR;
                2: CLASS↑.CLASS := SO;
                3: CLASS↑.CLASS := JR;
                4: CLASS↑.CLASS := SR;
            END;
            PUT(CLASS)
        END
    END.
```

Once we have our file of student records stored on disk, we will be wanting to insert grades from time to time. We could just type a list of grades, identified by SSN, for each homework assignment or exam and read the grades into the file by a program much like INITLOAD. But there are disadvantages to this approach. Some students may turn papers in late; others may argue (convincingly) that they deserve more points on one of the exams. We just can't count on being able to process all of the grades of a given type at the same time. A better way is to write an interactive program that asks the user:

1. Which student's record is to be updated
2. Which grade is to be entered
3. What the value of the grade is
4. Whether more grades are to be entered

There is a problem with writing this kind of program for sequential files. We can start at the beginning of a file (with RESET) and read it through, record by record. Or we can write records into the file one by one, as we did in INITLOAD. But we *cannot* read from and write to the same file at the same time. That is, we cannot read the record of a student, assign a value to a grade, and then write that record back to the file. What we have to do is to create a *second* file into which we write updated records (and copy the unchanged ones).

Since the processing of sequential files must proceed from the first record of the file through to the last, in writing our grade-entering program we shall assume that the records in the file CLASS are in order of increasing SSN, and that the user's list of grades to be entered is ordered in the same way. We shall also define a new external file called NEWCLAS (also made up of records of type STUDENT) into which we shall write the updated version of CLASS. Ignoring the checks for EOF, the program we shall write may be outlined as follows:

1. Prompt for SSN of student whose grade the user wishes to enter.
2. Proceed through the file CLASS, copying records to NEWCLAS until a matching SSN is found.
3. Prompt for the grades to be entered.
4. Write the updated record to NEWCLAS.
5. Repeat steps 1 through 4 until the user has no more grades to enter.
6. Copy the rest of CLASS to NEWCLAS.

Here, then, is the grade-entering program that we have written. Notice that this program has four external files as parameters. INPUT and OUTPUT (for interaction with the terminal), the file CLASS, which holds the existing version of the data (personal data plus previously entered grades), and the file NEWCLAS, to which we shall write the updated records.

```
PROGRAM ENTGRADE (INPUT/, CLASS, NEWCLAS, OUTPUT);
(* INSERT STUDENT TYPE DEFINITION HERE *)
VAR NEWCLAS, CLASS: FILE OF STUDENT;
    ANSWER, CONTINUE, NEXTGRADE: CHAR;
    S, N, I: INTEGER;
BEGIN
    WRITELN ('READY TO ENTER GRADES');
    WRITELN ('ANSWER YES/NO QUESTIONS WITH Y OR N');
    CONTINUE := 'Y';
    REWRITE(NEWCLAS);
    RESET(CLASS);
    WHILE CONTINUE = 'Y' DO
        BEGIN (* LOOP TO GET DATA FOR NEXT STUDENT *)
            WRITELN ('TYPE SSN OF STUDENT '); READLN;
            READ(S);
            (* SEARCH FILE FOR MATCHING SSN *)
```



```
WHILE NOT EOF(CLASS) AND (CLASS↑.SSN<>S) DO
  BEGIN
    NEWCLAS↑ := CLASS↑;
    PUT(NEWCLAS);          (* COPY RECORDS UNCHANGED *)
    GET(CLASS)
    END;
  IF EOF(CLASS) THEN WRITELN ('STUDENT NOT FOUND')
  ELSE BEGIN (* ACCEPTED VALID SSN *)
    NEXTGRADE := 'Y';
    WHILE NEXTGRADE = 'Y' DO (* PROMPT FOR GRADE *)
      BEGIN
        WRITELN ('ARE YOU ENTERING A HOMEWORK GRADE?');
        READLN;
        READ(ANSWER);
        IF ANSWER = 'Y'
        THEN (* PROMPT FOR HOMEWORK GRADE *)
          BEGIN
            WRITELN ('ENTER NUMBER OF ASSIGNMENT');
            READLN;
            READ(N);
            WRITELN ('ENTER GRADE'); READLN;
            READ (CLASS↑.HOMEWORK[N])
          END
        ELSE BEGIN (* IF NOT A HOMEWORK GRADE *)
            WRITE ('TYPE 1 FOR EXAM1, 2 FOR EXAM2, ');
            WRITELN ('TYPE 3 FOR FINAL GRADE');
            READLN;
            READ(I);
            WRITELN ('ENTER GRADE'); READLN;
            CASE I OF
              1: READ (CLASS↑.EXAM1);
              2: READ (CLASS↑.EXAM2);
              3: READ (CLASS↑.GRADE)
            END (* CASE *)
          END; (* ELSE *)
        WRITELN ('MORE GRADES FOR THIS STUDENT?');
        READLN;
        READ(NEXTGRADE)
      END (* LOOP FOR THIS STUDENT *)
    END; (* ACCEPTED VALID SSN *)
    CONTINUE := 'N';
    NEWCLAS↑ := CLASS↑;
    PUT(NEWCLAS);          (* WRITE UPDATED RECORD *)
    GET(CLASS);
    IF NOT EOF(CLASS) THEN BEGIN
      WRITELN ('DO YOU HAVE MORE GRADES?'); READLN;
      READ(CONTINUE)
```

```

                                END
                                END; (* LOOP TO GET DATA FOR NEXT STUDENT *)
WHILE NOT EOF(CLASS) DO BEGIN (* COPY REST OF CLASS *)
    NEWCLAS↑ := CLASS;
    PUT(NEWCLAS);
    GET(CLASS)
    END;
    WRITELN (' PROGRAM TERMINATING ')
END.

```

The two programs that we have written so far are used to enter data into the file. Next we need programs to retrieve data from the file. Retrieval programs may be classified into two types—*query programs* and *report-writing programs*. Query programs are interactive and allow users to look up information in a file. Report-writing programs generate printed reports of a specified format. Frequently several different report-writing programs are created to write different kinds of reports, but they may use data from the same file.

Any interactive query program for our file CLASS could look much like our program ENTGRADE. It would probably begin by asking which student the user is interested in and finding the record of that student. But then instead of asking which grade the user wants to enter and reading the value typed, it would ask what grade is needed and then would display that grade value on the terminal screen. Of course, since no changes are being made in the file, there is no need for the file NEWCLAS and the segments of code that write records to that file. As an exercise, make the necessary changes in program ENTGRADE to turn it into a query program.

Here we will give you two short examples of report-writing programs. The first, GRADES, prints a list of students (name and SSN), together with their final letter grades. The second, ENROLL, prints the total number of students in each class (FR, SO, JR, SR).

```

PROGRAM GRADES (CLASS, OUTPUT);
TYPE STUDENT = RECORD
    SSN: INTEGER;
    NAME: PACKED ARRAY[1..20] OF CHAR;
    CURRIC: 1000..9999;
    CLASS: (FR, SO, JR, SR);
    EXAM1: 0..100;
    EXAM2: 0..100;
    HOMEWORK: ARRAY[1..10] OF REAL;
    GRADE: CHAR
END;
VAR CLASS: FILE OF STUDENT;
BEGIN
    (* PRINT HEADINGS *)
    WRITELN ('GRADE REPORT FOR CS301, SPRING 82':41);
    WRITELN:

```

```
WRITE('SOCIAL SEC. NUMBER');  
Writeln('NAME':14, 'GRADE':17);  
Writeln;  
RESET(CLASS);  
WHILE NOT EOF(CLASS) DO BEGIN  
    Writeln(CLASS↑.SSN:13, CLASS↑.NAME:27,  
        CLASS↑.GRADE:7);  
    GET(CLASS)  
END  
END.
```

```
PROGRAM ENROLL (CLASS, OUTPUT);  
TYPE STUDENT = RECORD  
    SSN: INTEGER;  
    NAME: PACKED ARRAY[1..20] OF INTEGER;  
    CURRIC: 1000..9999;  
    CLASS: (FR, SO, JR, SR);  
    EXAM1: 0..100;  
    EXAM2: 0..100;  
    HOMEWORK: ARRAY[1..10] OF REAL;  
    GRADE: CHAR  
END;  
VAR CLASS: FILE OF STUDENT;  
    TOTAL: ARRAY[1..4] OF INTEGER;  
    TTOT, I: INTEGER;  
BEGIN  
    Writeln('FRESHMEN':14, 'SOPHOMORES':12,  
        'JUNIORS':10, 'SENIORS':10);  
    TTOT := 0;  
    FOR I := 1 TO 4 DO TOTAL[I] := 0;  
    RESET(CLASS);  
    WHILE NOT EOF(CLASS) DO BEGIN  
        CASE CLASS↑.CLASS OF  
            FR: TOTAL[1] := TOTAL[1] + 1;  
            SO: TOTAL[2] := TOTAL[2] + 1;  
            JR: TOTAL[3] := TOTAL[3] + 1;  
            SR: TOTAL[4] := TOTAL[4] + 1  
        END;  
        GET(CLASS)  
    END;  
    FOR I := 1 TO 4 DO  
        TTOT := TTOT + TOTAL[I];  
    Writeln(TOTAL[1], TOTAL[2]:12, TOTAL[3],  
        TOTAL[4])  
END.
```


15-8 EXERCISES

- 15-1.** Many programming languages include a facility for reading character strings represented in the INPUT textfile as a string of characters enclosed in single quotes, just as we represent character strings in a Pascal program. Write a procedure that scans a line of a textfile for such a character string and stores it in an array of 20 characters. If the string length is greater than 20, the string is to be truncated and its remaining characters discarded. If the string is less than 20 characters, it is to be padded with blanks (that is, the rest of the array is to be filled in with blanks).
- 15-2.** With reference to Exercise 15-1, character strings that extend over more than one line pose some interesting problems. Discuss these problems and how you might require the input to be typed to make handling such a string easier.
- 15-3.** Write a procedure similar to Example 15-5 that reads a REAL value.
- 15-4.** If you have done Exercises 15-1 and 15-3, you can combine them with Example 15-5 into a rather general READ procedure. To determine which procedure to call to read in the next item in the input list, you might maintain a table (e.g., an array of records) that gives the data type corresponding to each item name. (The system gets this information from variable declarations; you will have to supply it separately to your procedure.)
- 15-5.** Example 15-2 is the sort of program that obviously should be interactive. It is otherwise too tedious to remember the order in which the data items are to be typed. Write an interactive version of this program with nice prompt messages. Display an error message and give the user another chance if out-of-range data is entered. (Notice that then you do *not* want to use subrange data types as we did.)
- 15-6.** Unlike some programming languages, Pascal has no built-in facility for handling complex numbers. A convenient way to work with complex numbers in Pascal is to define a new type:

```
TYPE COMPLEX = RECORD
    REALPART, IMAGPART: REAL
END;
```

Write procedures to multiply and add complex numbers. The parameters should be three variables of type COMPLEX—the two operands and the result. Incorporate these procedures into an interactive program that does complex arithmetic, asking the user for two operands and an operator ('+', '-', '/', or '*'), and displaying the real and imaginary parts of the result.

SUMMARY OF CHAPTER 15

RECORD is a structured type in which the components, called fields, may be of different types. A field of a record may itself be structured, e.g., another RECORD type.

Groups of records of the same type may be stored in arrays or in files. FILE is a structured type in which all components are of the same type; but, unlike array components, the components of a file are not numbered. Files can have components of any defined type (except another file). Files are generally stored on an auxiliary storage device (such as a magnetic disk), and transferred to main memory one component at a time. Thus, large amounts of data are usually stored as files of records, and not as arrays.

Pascal files are sequential; successive components can only be fetched from auxiliary memory in order, starting from the first component. Special Pascal file-handling commands are PUT (to write to a file), GET (to read from a file), RESET (to start at the top of a file), and REWRITE (to clear an old file prior to putting new information in it.)

16

POINTERS AND LISTS

16-1 INTRODUCTION

Probably you have all played some form of a "treasure hunt" game. In a typical treasure hunt, a troop of Boy Scouts are set to work tracking down clues to the solution of a puzzle, perhaps the location of a "treasure." To begin, the Scouts are given a first clue, together with an instruction such as "Go to the big tree in front of the haunted house for the next clue." With the second clue might be the further instruction, "Go to the covered bridge for the next clue." At the covered bridge a Scout would then find both the third clue and information on the location of the fourth clue. In this way, one clue leads to the next, as depicted in Figure 16-1. The arrows show how the Scouts are directed from one clue to the next.

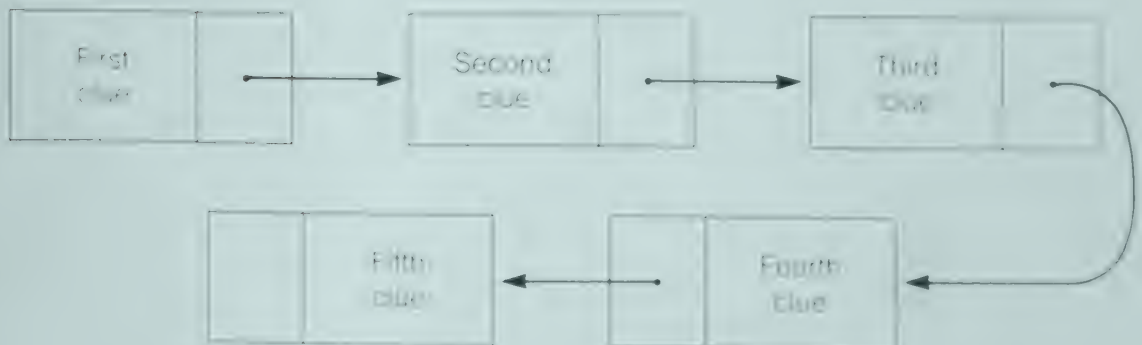


Figure 16.1

We are interested in the way these clues are organized. In particular, we observe that:

1. The troop leader who sets up the clues might not know at the beginning how many clues he wishes to have. He might simply set up the first clue and the instruction as to how to get to the next clue. He can then later set up a second clue with directions for finding a third clue. In short, the clues can be set up dynamically, one at a time.
2. Suppose that after all the clues are set up, the troop leader decides to delete a clue, say the fourth one. He can do so by making one simple change, namely, by changing the directions to the next clue that are given with the third clue from the location of the fourth clue to the location of the fifth. The Scouts will then go directly from the third clue to the fifth, skipping the fourth. Figure 16-2 shows the situation. (Note that if the leader is tired he need not retrieve the old fourth clue—as long as none of the Scouts is likely to get lost and stumble upon it accidentally.)
3. On the other hand, suppose that the troop leader wishes to add a new clue between the third and fourth clues. In this case, he needs to give the directions to the new clue with the third clue and include with the new clue the directions to the old fourth clue. Figure 16-3 shows the new situation.

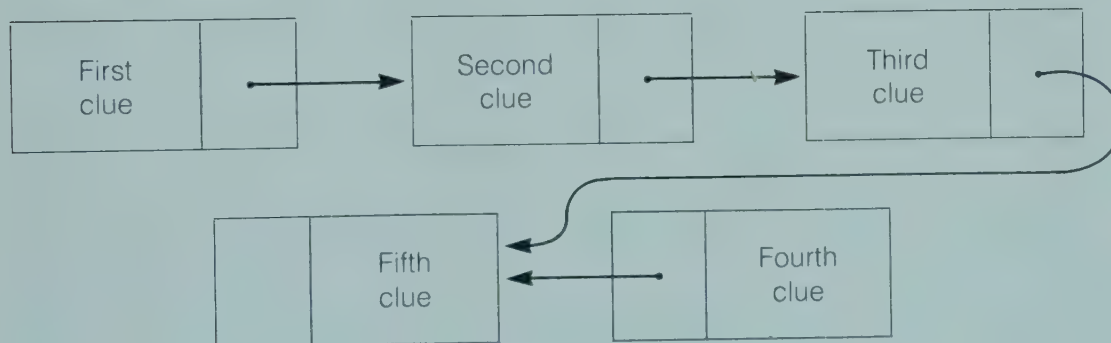


Figure 16.2

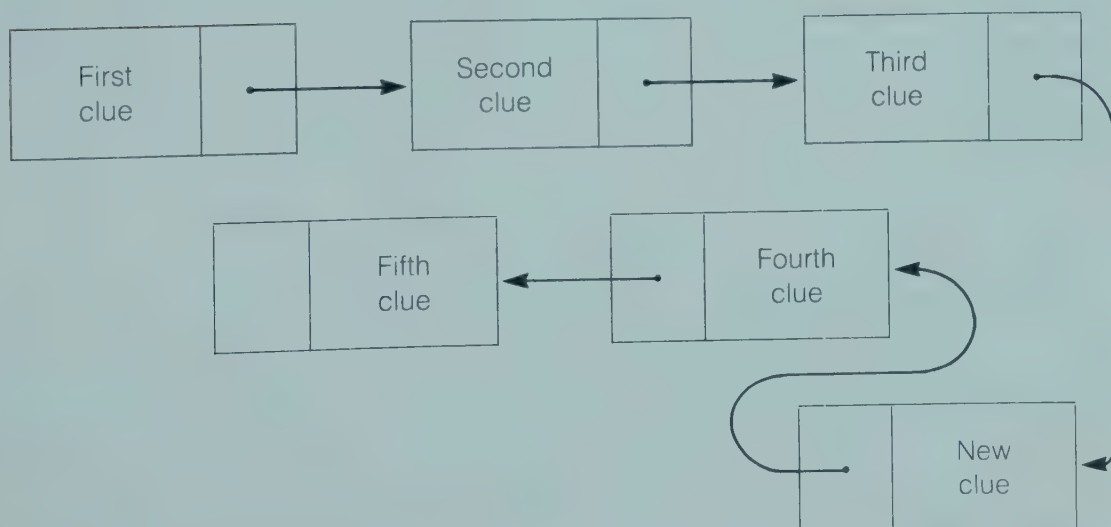


Figure 16.3

Now let us look at a computing analogy. Suppose that we have a program for editing text material. How do we store a sentence in the computer? One way is to use an array, storing each word of the sentence as one entry in the array. For example, we might have

```
TEXT[1] = 'GO   '  
TEXT[2] = 'TO   '  
TEXT[3] = 'THE  '  
TEXT[4] = 'RED  '  
TEXT[5] = 'BRICK'  
TEXT[6] = 'HOUSE'
```

Now suppose we want to change the sentence to GO TO THE BRICK HOUSE. We need to delete the word RED and move the subsequent words up, so that

```
TEXT [4] = 'BRICK '  
TEXT [5] = 'HOUSE '  
TEXT [6] = '      '
```

On the other hand, suppose that we want to change the sentence to GO TO THE OLD RED BRICK HOUSE. We need to insert the word OLD and move the subsequent words down so that

```
TEXT [4] = 'OLD   '  
TEXT [5] = 'RED   '  
TEXT [6] = 'BRICK '  
TEXT [7] = 'HOUSE'
```

Clearly this way of storing text is very inconvenient, since minor changes cause much moving of words. Indeed, the idea of going from one clue to another that we learned from the treasure hunt is just what we need. Imagine that the words in a sentence are chained together by arrows, as shown in Figure 16-4. The deletion of the word RED or the insertion of the word OLD can be handled by changing one or two arrows, without disturbing the other arrows, as shown in Figure 16-5a and b. Notice that the changes in the arrows are completely analogous to the changes in directions that the Scout leader has to make to effect an insertion or deletion in the sequence of clues.

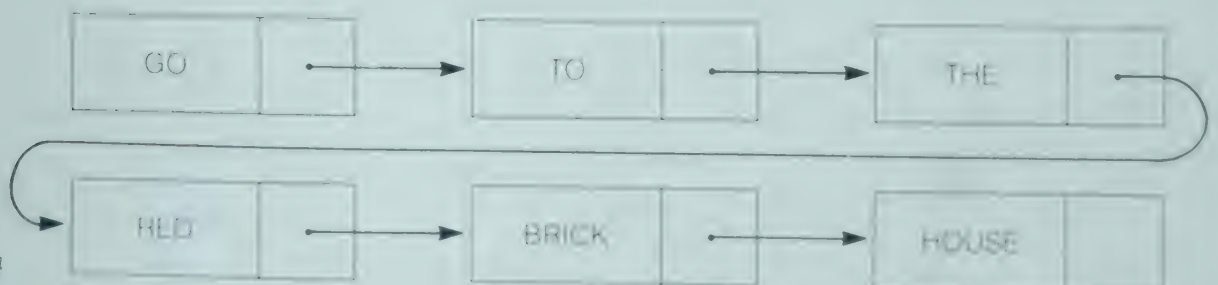
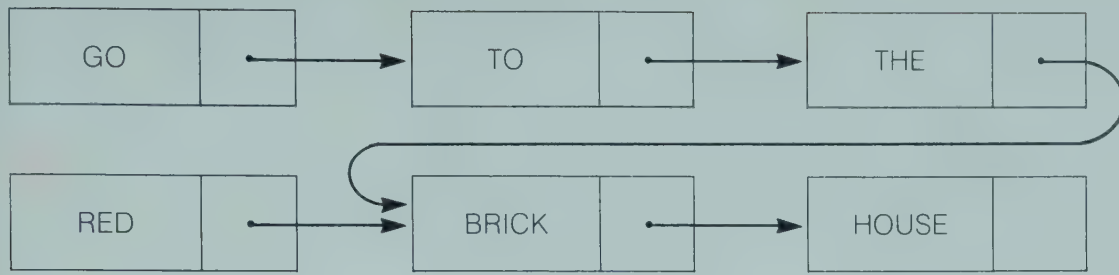
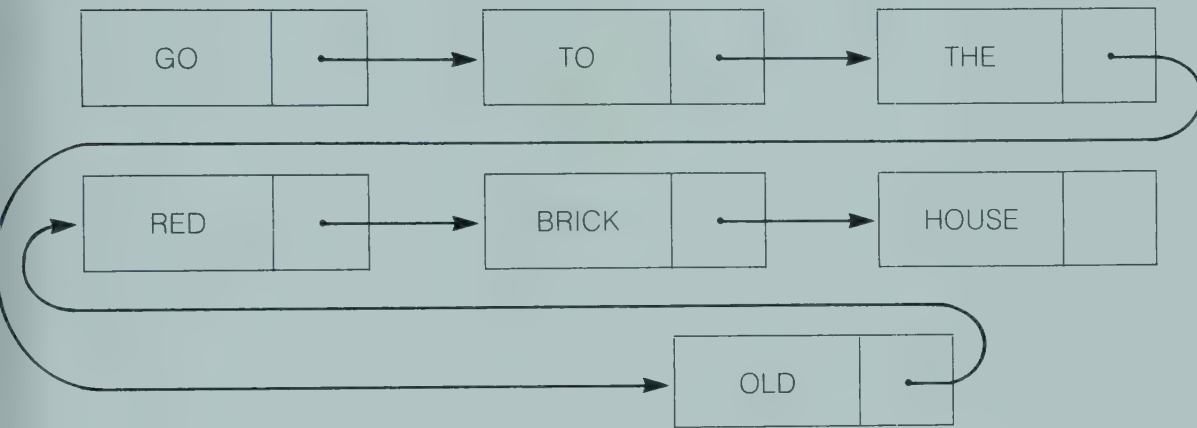


Figure 16.4



(a)



(b)

Figure 16.5

It turns out that in maintaining data in a computer there are many occasions on which we have data items that we wish to chain together in this way. Our question now is how can we chain items together? In particular, how can we set up “arrows” leading from one data item to the next? The Pascal language provides us with such “arrows”; they are called *pointers*. Pointers are just another data type, as we shall discuss in the next section.

16-2 POINTERS

We now introduce the data type *pointer*. Pointer is a simple type, similar to INTEGER, BOOLEAN, and REAL. Pointer variables are used to point to variables of other types. Thus a pointer variable is not simply declared as being of type POINTER, but is declared as a type that is a pointer to a particular data type. For example, the definition

```
TYPE LINK = ↑CHAR;
```


states that variables of type LINK are pointers to variables of type CHAR. Thus we might have

```
TYPE LINK = ↑CHAR;  
      CHAIN = ↑INTEGER;  
VAR P,Q: LINK;  
      R,S: CHAIN;
```

Here P and Q are of the type LINK. That is, they are pointers pointing at (or referencing) CHAR variables. Also, R and S are variables of the type CHAIN and thus are pointers referencing INTEGER variables. As usual, we needn't have defined the types LINK and CHAIN but could simply have declared

```
VAR P,Q: ↑CHAR;  
      R,S: ↑INTEGER;
```

In any case, the situation that is set up by these declarations can be pictured as in Figure 16-6. At this moment, the pointers P, Q, R, and S exist, and we have declared what type of variable each may point to. But they are not yet pointing at any variables.

How then do we arrange for a pointer to point at a variable? One way you might think of is to declare a variable of the proper type and somehow set the value of the pointer so that it points to the variable. However, we cannot do this in Pascal. What we do in Pascal is to use a standard procedure named NEW to create the variables to which the pointers point. Thus the statements

```
NEW(P);  
NEW(Q);  
NEW(R);  
NEW(S)
```

will set up memory locations for four variables of the correct types and assign values to the pointers P, Q, R, and S so that they point to these newly created variables. We then have the situation shown in Figure 16-7.

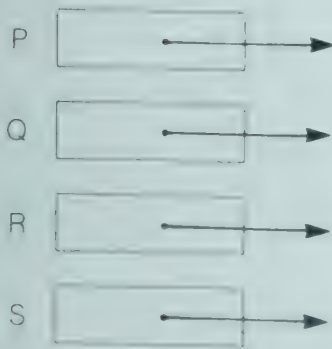


Figure 16.6

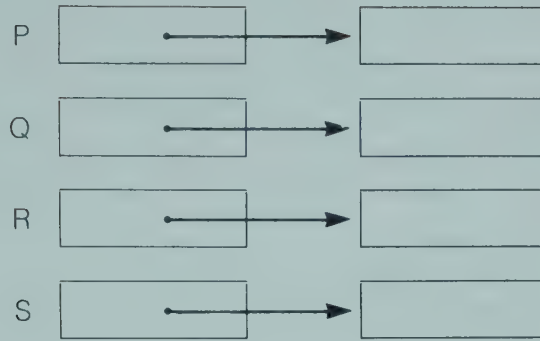


Figure 16.7

Next we need to know how these new variables are referenced. The answer is that the variables that P , Q , R , and S point to are referred to as $P \uparrow$, $Q \uparrow$, $R \uparrow$, and $S \uparrow$, respectively. Clearly, $P \uparrow$ and $Q \uparrow$ are of type CHAR, and $R \uparrow$ and $S \uparrow$ are of type INTEGER. Therefore, if we write

```
P↑ := 'G';
Q↑ := '0';
R↑ := 47
```

you should have no trouble understanding what this means. Similarly, we can write

```
P↑ := Q↑
```

or

```
S↑ := R↑ + 1
```

since an INTEGER variable referenced by a pointer can be used in expressions just like any other INTEGER variable. Furthermore, the statement

```
P := Q
```

causes P to point to the same variable that Q points to. Thus pictorially we would have the situation shown in Figure 16-8.

In the following example (Example 16-1), we first declare an array P of pointers to INTEGER variables. We then read integer values into these variables, creating the variables dynamically as we input the data. The situation after the data is read might then be as shown in Figure 16-9. Then we rearrange the pointers so that $P[1] \uparrow < P[2] \uparrow < \dots < P[10] \uparrow$. To accomplish

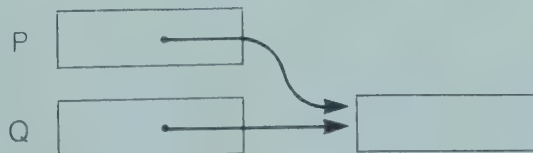


Figure 16.8

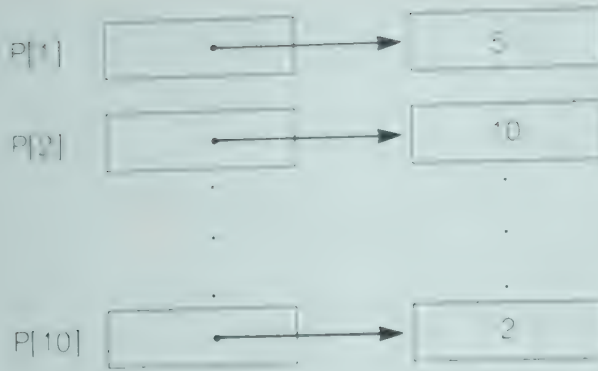


Figure 16.9

this task we use the simple bubblesort algorithm, but, instead of rearranging the integers themselves, we merely rearrange the pointers. This is a useful technique to remember, especially if the items to be sorted are lengthy (e.g., RECORD types).

EXAMPLE 16-1

```

PROGRAM PTRSORT ( INPUT, OUTPUT );
CONST N = 10;
TYPE LINK = ↑INTEGER;
VAR P: ARRAY[1..N] OF LINK;
    TEMP: LINK;
    I, J: INTEGER;
BEGIN
  FOR I := 1 TO N DO
    BEGIN (* READ INTEGERS *)
      NEW(P[I]);
      READ(P[I]↑);
    END;
  (* NOW SORT THE INTEGERS *)
  FOR I := 1 TO N - 1 DO
    FOR J := 1 TO N - I DO
      IF P[J]↑ > P[J+1]↑ THEN
        BEGIN (* EXCHANGE *)
          TEMP := P[J];
          P[J] := P[J+1];
          P[J+1] := TEMP;
        END;
    END;
  (* WRITE THE SORTED INTEGERS *)
  FOR I := 1 TO N DO WRITELN(P[I]↑);
END.

```

16-3 LINKED LISTS

Let us return to the example in Section 16-1. Recall that we had a sequence of items in which one follows another. In particular, each item consisted of a

clue and an associated pointer to the next item. Such a data structure can be set up as in the following example.

EXAMPLE 16-2

```
PROGRAM CLUES (INPUT, OUTPUT);
TYPE LINK = ↑ITEM;
   CLUETYPE = PACKED ARRAY[1..20] OF CHAR;
   ITEM = RECORD
       CLUE: CLUETYPE;
       NEXT: LINK
   END;
VAR FIRST, PTR: LINK;

PROCEDURE READSTRING(VAR STRING: CLUETYPE);
(* READS A CLUE AND PACKS IT INTO STRING *)
VAR I: INTEGER;
    CHARS: ARRAY[1..20] OF CHAR;
BEGIN
    FOR I := 1 TO 20 DO
        READ(CHARS[I]);
    READLN;
    PACK(CHARS, 1, STRING)
END;

PROCEDURE PRINTCLUES;
BEGIN
    WRITELN(FIRST↑.CLUE);
    PTR := FIRST;
    WHILE PTR↑.NEXT <> NIL DO
        BEGIN
            PTR := PTR↑.NEXT;
            WRITELN(PTR↑.CLUE)
        END
    END;
END;

BEGIN (* MAIN PROGRAM: READ CLUES, BUILD LIST *)
    NEW(FIRST);
    READSTRING(FIRST↑.CLUE);
    PTR := FIRST;
    WHILE PTR↑.CLUE <> 'I HAVE NO MORE CLUES' DO
        BEGIN
            NEW(PTR↑.NEXT);
            PTR := PTR↑.NEXT;
            READSTRING(PTR↑.CLUE)
        END;
    PTR↑.NEXT := NIL;
    PRINTCLUES
END.
```

There are several points to notice about this example.

1. The way we have associated each clue with a pointer to the next is to define the items as records, with one field a character string to hold the clue and the other field a pointer to point to the next item.
2. We have defined two variables of type LINK. The one named FIRST always points to the beginning of the list. The other (named PTR) is assigned the successive pointer values as they are created (or, in PRINTCLUES, as the list is traversed and printed).
3. After the last clue message (I HAVE NO MORE CLUES) is read, we indicate the end of the list by assigning the value NIL to PTR↑.NEXT. NIL, a Pascal reserved word, denotes a special value that may be assigned only to pointers. A value of NIL means that the pointer is pointing nowhere; it is conveniently used to end a list. A procedure that later reads the list can then test for the end by looking for the condition PTR = NIL.
4. The type ITEM is defined *after* the type ↑ ITEM. This is a rare exception to the general Pascal rule that names must be defined before they are used.
5. Like files, lists are indefinitely expandable. If we want to add another item, we simply call upon the procedure NEW to create space for it.
6. We have included the procedure PRINTCLUES to illustrate how simple it is to find one's way through a linked list by following the pointers.

In our discussion we have implied that the advantage of maintaining the clues in a linked list is that it is easy to delete old clues or to insert new clues. Here are program segments to carry out these operations. Notice that in Example 16-3 (the segment that deletes a clue) we introduce a new standard procedure, DISPOSE, that returns the memory space containing the deleted record to the free space available to our program. If we make many changes in a linked list and never utilize DISPOSE, we will eventually run out of space for new records. The procedure NEW would then not be able to find empty space and would return an error message.

EXAMPLE 16-3

```
( * TO DELETE GIVENCLUE * )
PTR := FIRST;
WHILE ( PTR↑.CLUE<>GIVENCLUE ) AND ( PTR↑.NEXT<>NIL )
DO BEGIN
    ( * SEARCH LIST FOR GIVENCLUE * )
    TEMP := PTR; ( * SAVE PREVIOUS POINTER * )
    PTR := PTR↑.NEXT
END;
IF PTR↑.CLUE<> GIVENCLUE THEN
    WRITELN( 'GIVENCLUE NOT FOUND ' )
ELSE
```

```

TEMP↑.NEXT := PTR↑.NEXT;
( * CHANGE PREVIOUS POINTER TO SKIP THIS
  ITEM * )
DISPOSE(PTR)

```

EXAMPLE 16-4

```

( * TO ADD NEW CLUE AFTER GIVENCLUE * )
PTR := FIRST;
WHILE(PTR↑.CLUE <> GIVENCLUE)
  AND(PTR↑.NEXT<>NIL) DO
  ( * SEARCH LIST FOR GIVENCLUE * )
    PTR := PTR↑.NEXT;
IF PTR↑.CLUE<>GIVENCLUE THEN
  WRITELN('GIVENCLUE NOT FOUND')
ELSE BEGIN
  TEMP := PTR↑.NEXT;
  NEW(PTR↑.NEXT);
  PTR := PTR↑.NEXT;
  PTR↑.CLUE := NEWCLUE;
  PTR↑.NEXT := TEMP
END;

```

Study these two program segments. Draw items with pointers and follow through the changes to see how it all works. Does the deletion segment correctly handle the special cases where the GIVENCLUE appears either first or last in the list? (If not, make the appropriate changes.) The insertion segment is somewhat special-purpose, in that we have assumed that we will not want to insert a new first clue, nor append a new clue after the one that reads I HAVE NO MORE CLUES. A more general list-insertion procedure would allow one to link in new items at the beginning or end of a list, as well as in the middle.

In the examples above, NEW is always used to allocate space for records without variant parts and hence of a fixed, known length. If you want to create new records of a type that has variant parts, you must tell the procedure NEW which variant (or variants, in case of nesting) you are requesting space for. You do this by including in the actual parameters for NEW a list of the tag-field values, in the order in which they were declared. The format is, for example,

```
NEW(POINTER, TAG1-VALUE, TAG2-VALUE)
```

for a record with two variants. To DISPOSE of such a record, you would have to write

```
DISPOSE(POINTER, TAG1-VALUE, TAG2-VALUE)
```


where the tag values would, of course, have to exactly match those of the record.

A final note: the use of pointers allows us to link records in arbitrary patterns, perhaps in a circle as shown in Figure 16-10. Or into a treelike structure (with the "root" at the top, so that it looks upside down). See Figure 16-11. Notice that in this figure each record contains two pointers, one to the right "branch" and one to the left "branch." This is perfectly legal.

Discussion of these and other more complicated "data structures" is beyond the scope of this text. Whole books exist on this topic.

16-4 SYNTAX

We have already seen that the construct



can legally be used for a *variable*. In Chapter 15, the variable identifier was a file name; here it is a pointer name. The usages are not really so different; indeed we can think of a file window as being a pointer to a component of the file.

Thus the only new syntax in this chapter is a final addition to *type*:

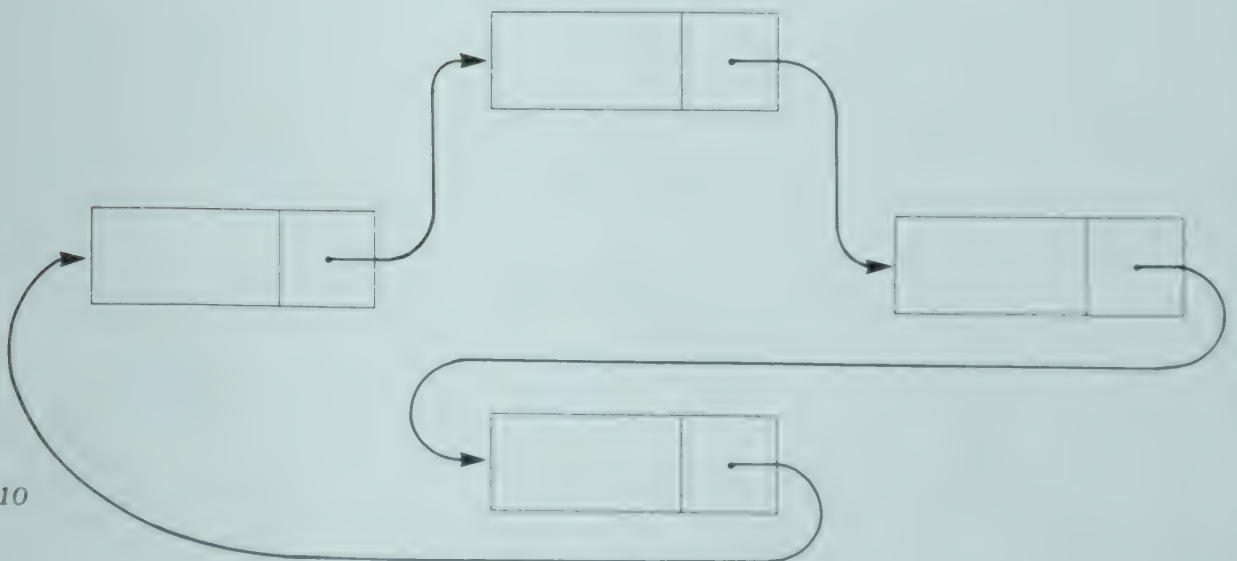
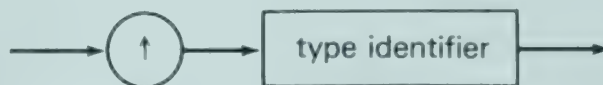


Figure 16.10

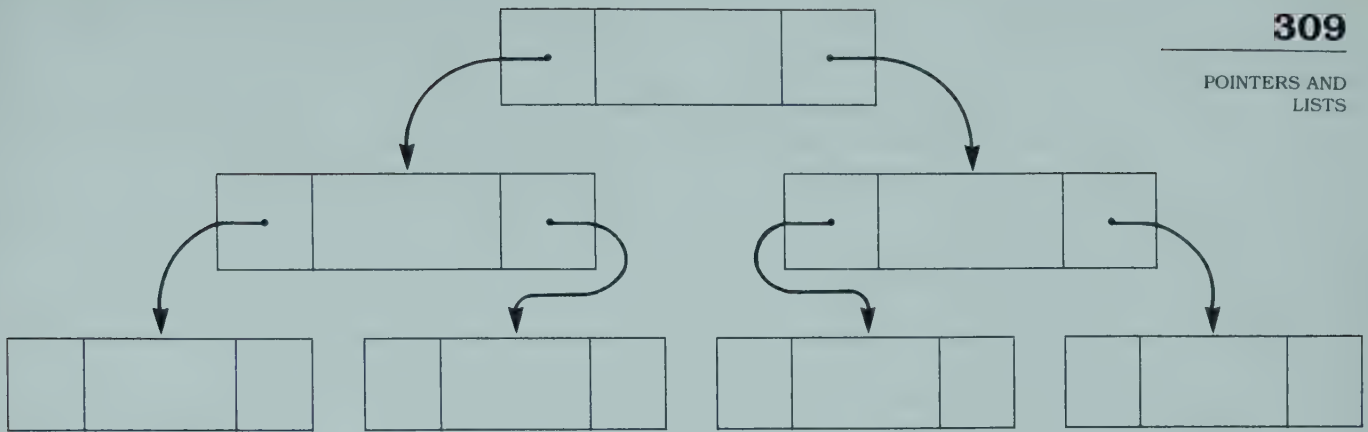


Figure 16.11

16-5 A PROGRAMMING EXAMPLE: EDITING TEXT

In Section 16-1 we discussed the possibility of storing sentences in the computer as arrays of words. Because of the problems of inserting and deleting words, we now see that a much better strategy would be to maintain linked lists of words. But this still isn't good enough. For one thing, each of the items in a linked list must be of the same type; thus all words would have to be of the same length. We would then define a "word" to be of the maximum possible length and pad the words we enter with blanks so that they fit. This clearly is a nuisance, not to mention being wasteful of space. The other problem is that we may wish to change the spelling of a word, and fixing just one or two letters would necessitate working with the whole word. A much more flexible approach is to store a sentence as a linked list of characters. This is the approach that we shall discuss here.

To simplify matters, our program will include the input and editing of only a single sentence. We shall make the program interactive, as a text editor should be. And we shall include only one editing command, a "substitute" or "change" command, which we shall call CORRECT, to allow the user to substitute one character string for another within the sentence. The overall structure of the program will then be something like that shown in Figure 16-12. We could include more commands in this structure by converting the branch that tests whether the command is a 'C' to a CASE construct that results in a call to one of various procedures, depending upon the command that the user has typed.

CORRECT is easily implemented by the procedure outlined in Figure 16-13. The original sentence, the string to be replaced (OLDSTRING), and the string to be inserted (NEWSTRING) in place of OLDSTRING are all stored in linked lists for easy manipulation. The most complicated part of the correction process is locating the string to be replaced within the original sentence. To do this we have written a procedure called MATCH that finds the first occurrence of a given string S1 in a second given string S2. Pointers to these strings are parameters of MATCH. MATCH then returns the information that

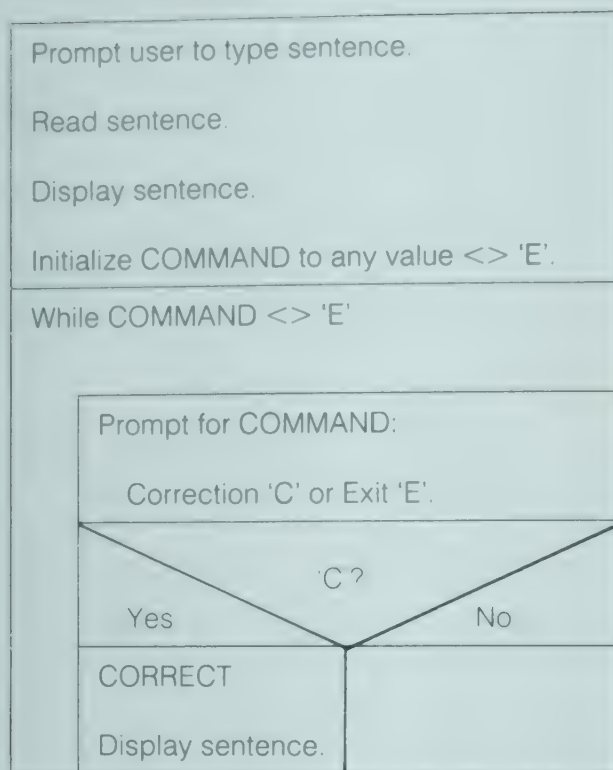
Main Procedure

Figure 16.12

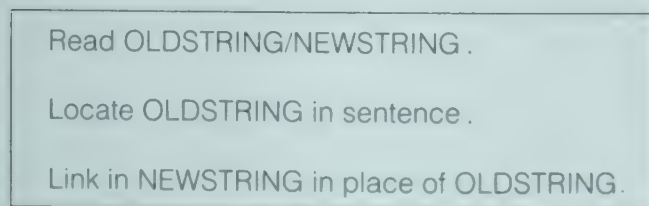
Correct

Figure 16.13

CORRECT needs to link in NEWSTRING; namely, a pointer to the character that NEWSTRING is to follow and a pointer to the first character of the string to be linked in at the end of NEWSTRING. If a change is to be made at the beginning of the sentence, there is no old character that NEWSTRING is to follow. MATCH returns a BOOLEAN value (BEGINLIST) to indicate when this situation has occurred.

The general outline of MATCH is given in Figure 16-14. The test for whether a match was found is simply whether the end of S1 was reached before a mismatch was found. To implement "Go back and try again", the pointers to S1 and S2 are reset to the beginning of S1 and to the next character after the previous starting point in S2, respectively.

With this much introduction, you are ready to study Example 16-5.

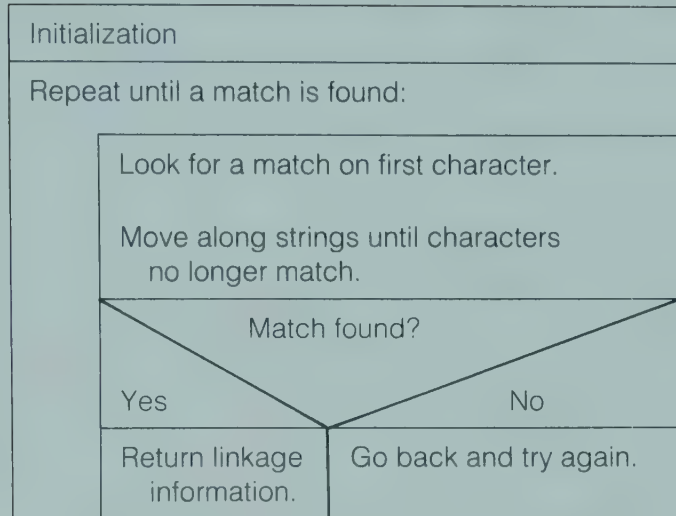


Figure 16.14

EXAMPLE 16-5

```

PROGRAM EDIT(INPUT/,OUTPUT);
TYPE LINK = ↑CHARACTER;
      CHARACTER = RECORD
          CH : CHAR;
          PTR: LINK
      END;
VAR FIRST, THIS: LINK;
    COMMAND: CHAR;

PROCEDURE PRINTSTRING(THIS: LINK);
(* PRINTS THE SENTENCE *)
BEGIN
    WRITE(THIS↑.CH);
    WHILE THIS↑.PTR <> NIL DO
        BEGIN
            THIS := THIS↑.PTR;
            WRITE(THIS↑.CH)
        END;
    WRITELN
END;

PROCEDURE READSTRING(TERMINAL: CHAR; VAR TEMP: LINK);
(* READ A STRING ENDING WITH A SPECIFIED TERMINAL CHAR *)
(* PUT IT IN LIST POINTED TO BY TEMP *)
VAR THIS: LINK;
BEGIN
    NEW(TEMP);

```

```

READ(TEMP↑.CH);
THIS := TEMP;
WHILE THIS↑.CH <> TERMINAL DO BEGIN
    NEW(THIS↑.PTR);
    THIS := THIS↑.PTR;
    READ(THIS↑.CH)
END;
THIS↑.PTR := NIL
END;

```

```

PROCEDURE MATCH(S1,S2: LINK;
                VAR INS, OUTS: LINK;
                VAR BEGINLIST: BOOLEAN);
(* FINDS FIRST OCCURRENCE OF STRING S1 IN STRING S2 *)
VAR SUBSTRING, TEMP: LINK;
    FOUND: BOOLEAN;
BEGIN
    FOUND := FALSE;
    INS := S2;
    BEGINLIST := TRUE;
    OUTS := NIL;
    SUBSTRING := S2;
    WHILE FOUND = FALSE DO BEGIN
        WHILE SUBSTRING↑.CH <> S1↑.CH DO BEGIN
            BEGINLIST := FALSE;
            INS := SUBSTRING;
            SUBSTRING := SUBSTRING↑.PTR
        END;
        TEMP := S1;
        WHILE (SUBSTRING↑.CH = TEMP↑.CH) AND
            (TEMP↑.PTR <> NIL) DO BEGIN
            SUBSTRING := SUBSTRING↑.PTR;
            TEMP := TEMP↑.PTR
        END; (* LOOP TO MATCH *)
        IF TEMP↑.PTR = NIL THEN (* MATCH WAS FOUND *)
            BEGIN
                OUTS := SUBSTRING;
                FOUND := TRUE
            END
        ELSE BEGIN (* TRY AGAIN *)
            IF BEGINLIST THEN
                SUBSTRING := INS↑.PTR
            ELSE BEGIN
                INS := INS↑.PTR;
                SUBSTRING := INS↑.PTR
            END;
            BEGINLIST := FALSE
        END
    END

```

```

        END (* TRY AGAIN *)
    END (* WHILE LOOP THAT SEARCHES *)
END; (* PROCEDURE MATCH *)

(*
*)
(* THE NEXT PROCEDURE CONTROLS THE
*)
(* CORRECTION PROCESS
*)
*)
PROCEDURE CORRECT;
VAR OLDSTRING, NEWSTRING, LINKIN, LINKOUT, THIS: LINK;
    C: CHAR;
    BEGINLIST: BOOLEAN;
BEGIN
    READ(C);
        (* READ THE / *)
    READSTRING('/', OLDSTRING);
    MATCH (OLDSTRING, FIRST, LINKIN, LINKOUT, BEGINLIST);
    READSTRING('/', NEWSTRING);
    IF BEGINLIST THEN
        FIRST := NEWSTRING
    ELSE LINKIN↑.PTR := NEWSTRING;
    (* NEXT FIND END OF NEWSTRING *)
    THIS := NEWSTRING;
    WHILE THIS↑.PTR↑.PTR <> NIL DO
        THIS := THIS↑.PTR;
    (* NOW LINK IN THE REST OF THE OLD STRING *)
    THIS↑.PTR := LINKOUT;
    PRINTSTRING(FIRST)
END; (* PROCEDURE CORRECT *)

BEGIN (* MAIN PROGRAM *)
    WRITELN('TYPE A SENTENCE. ');
    WRITELN('MAKE SURE IT ENDS WITH A PERIOD. ');
    READLN;
    READSTRING('.', FIRST);
    PRINTSTRING(FIRST);
    COMMAND := 'A';
    WHILE COMMAND <> 'E' DO BEGIN
        WRITELN('DO YOU WANT TO CORRECT THIS SENTENCE? ');
        WRITELN('IF SO, TYPE C/OLDSTRING/NEWSTRING/ ');
        WRITELN('IF NOT, TYPE E TO EXIT. ');
        READLN;
        READ(COMMAND);
        IF COMMAND = 'C' THEN CORRECT
    END
END.

```


16-6 EXERCISES

- 16-1.** What is the output of the following procedure when it is called by
 (a) ALPHABET(26)?
 (b) ALPHABET(5)?

```

PROCEDURE ALPHABET(J: INTEGER);
TYPE LINK = ↑CHARACTER;
      CHARACTER = RECORD
                  CH: CHAR;
                  PTR: LINK
                END;
VAR FIRST, THIS: LINK; I: INTEGER;
BEGIN
  NEW(FIRST);
  THIS := FIRST;
  FOR I := 1 TO J DO BEGIN
    NEW(THIS↑.PTR);
    THIS := THIS↑.PTR
  END;
  THIS↑.PTR := FIRST;
  J := ORD('A');
  THIS := FIRST;
  FOR I := J TO J + 25 DO BEGIN
    THIS↑.CH := CHR(I);
    THIS := THIS↑.PTR
  END;
  THIS := FIRST;
  FOR I := 1 TO 26 DO BEGIN
    WRITE(THIS↑.CH);
    THIS := THIS↑.PTR
  END
END;

```

- 16-2.** A realistic text editor must be written to handle virtually any kind of user error. In particular, code to handle a string substitution command, like that in Example 16-5, cannot assume that the user will correctly type an OLDSTRING that is a substring of the sentence to be corrected. Alter procedure MATCH so that it returns NIL for the value of INS if the substring is not found. Then make appropriate changes in the rest of the program to print an error message and give the user another chance to type the command correctly.
- 16-3.** With reference to Exercise 16-2, another error that the user might make is to forget the terminal slash (/). How would you handle this problem? Does the "substitute" command in the text editor on your computer require such a terminal character? What problems can arise without one?

- 16-4.** Is the CORRECT command in Example 16-5 adequate for:
- (a) Deleting any portion of the sentence?
 - (b) Inserting new material in the middle of the sentence?
 - (c) Inserting new material before the beginning of the current sentence?
 - (d) Appending new material at the end of the sentence?
 - (e) Replacing the whole sentence by a completely new sentence?
- 16-5.** Example 16-5 illustrates a situation where DISPOSE should be used to keep old, discarded strings from taking up valuable memory space. Write a procedure GIVEBACK that releases the space occupied by OLDSTRING. Indicate at what point in Example 16-5 GIVEBACK would be called.

SUMMARY OF CHAPTER 16

Pointers may be defined to point to items of any Pascal data type. A pointer type definition looks like

TYPE name = ↑ type;

The variable pointed to by a variable of type pointer is referenced as

pointername ↑

The standard procedure NEW allocates space for a variable pointed at by a pointer. The standard procedure DISPOSE can be used to release that space when it is no longer needed. By defining pointers that point to records that contain a field of that pointer type, one can create linked lists of data items.

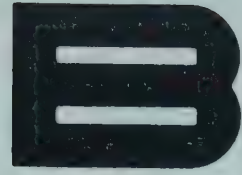
A

APPENDIX

LIST OF PASCAL RESERVED WORDS

Note: Some systems may not recognize all the reserved words listed below. Other systems may have other reserved words added to their versions of Pascal. Programmers will also wish to avoid using for other purposes the standard constants (TRUE, FALSE, MAXINT), the names of the standard data types (INTEGER, BOOLEAN, REAL, CHAR, TEXT), the standard program parameters INPUT and OUTPUT, and the names of the standard functions and procedures implemented on their systems.

AND	NIL
ARRAY	NOT
BEGIN	OF
CASE	OR
CONST	PACKED
DIV	PROCEDURE
DO	PROGRAM
DOWNTO	RECORD
ELSE	REPEAT
END	SET
FILE	THEN
FOR	TO
FUNCTION	TYPE
GOTO	UNTIL
IF	VAR
IN	WHILE
LABEL	WITH
MOD	

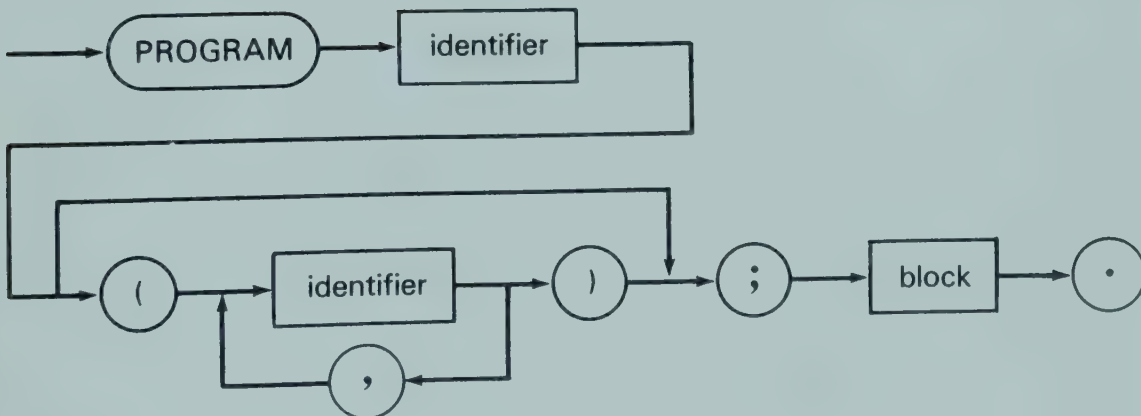


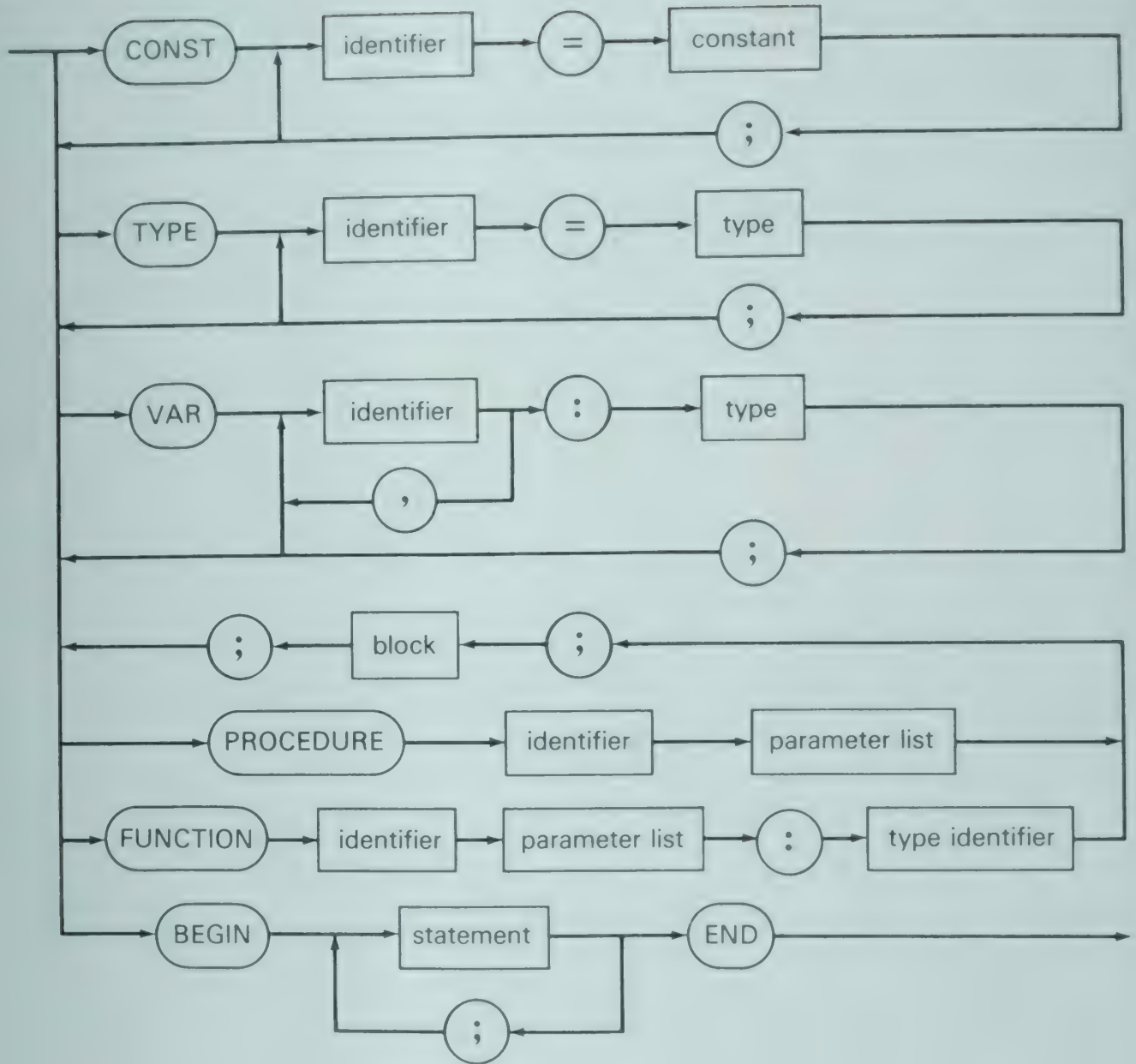
APPENDIX

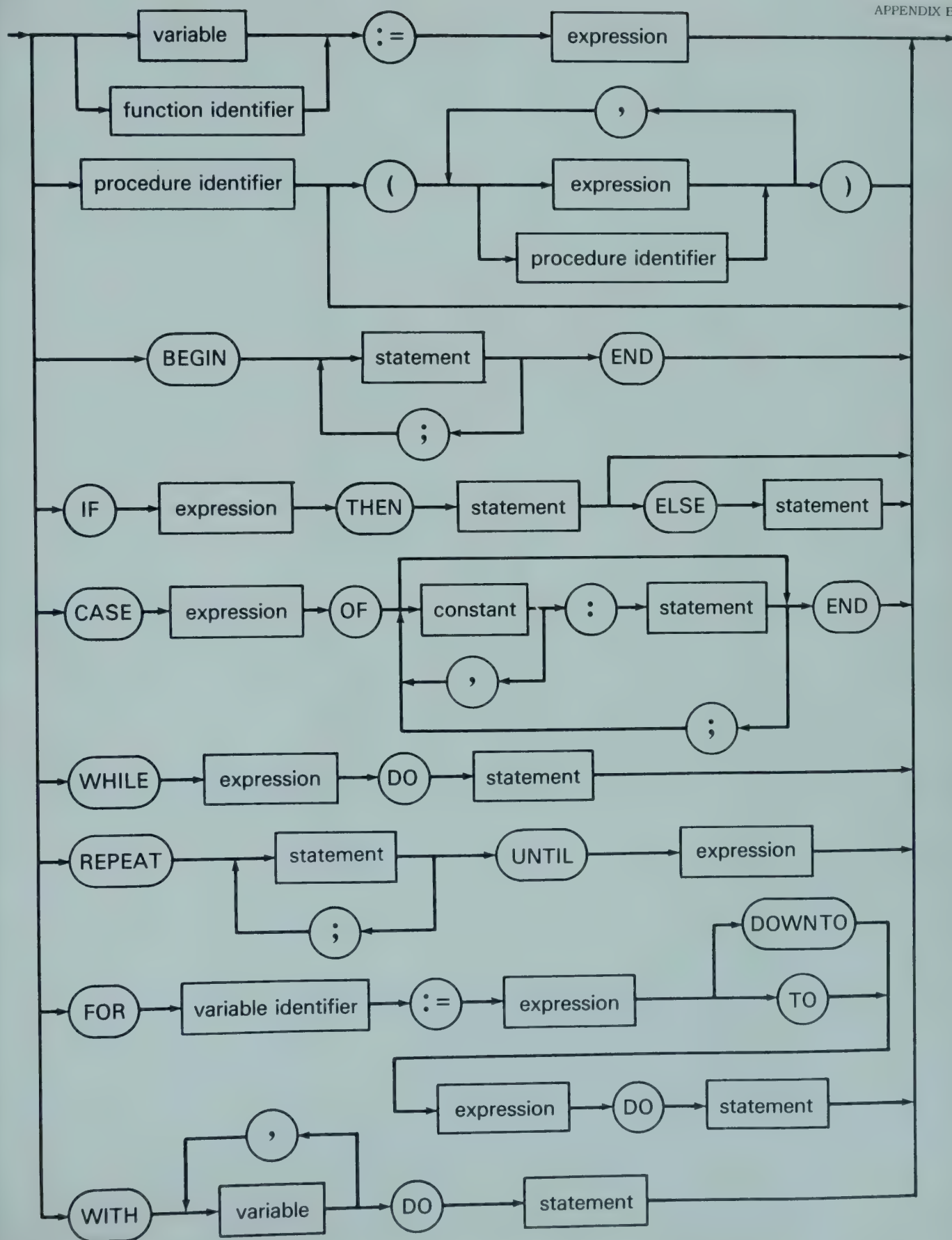
SYNTAX DIAGRAMS FOR PASCAL

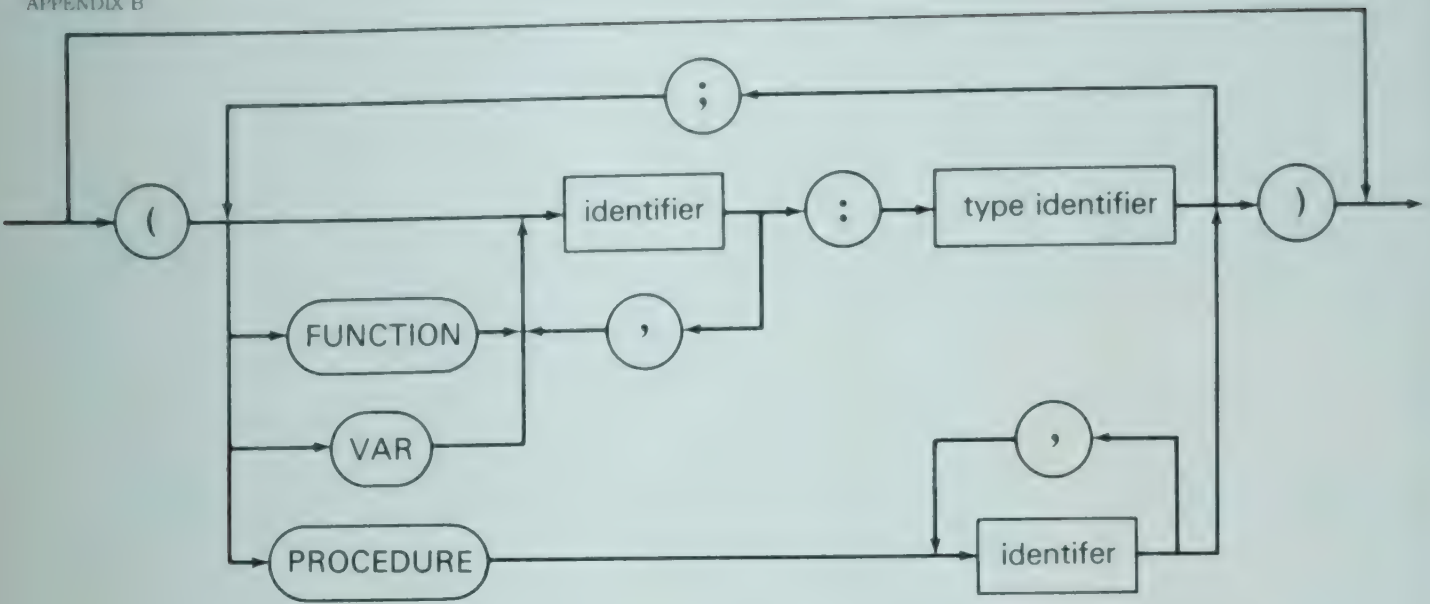
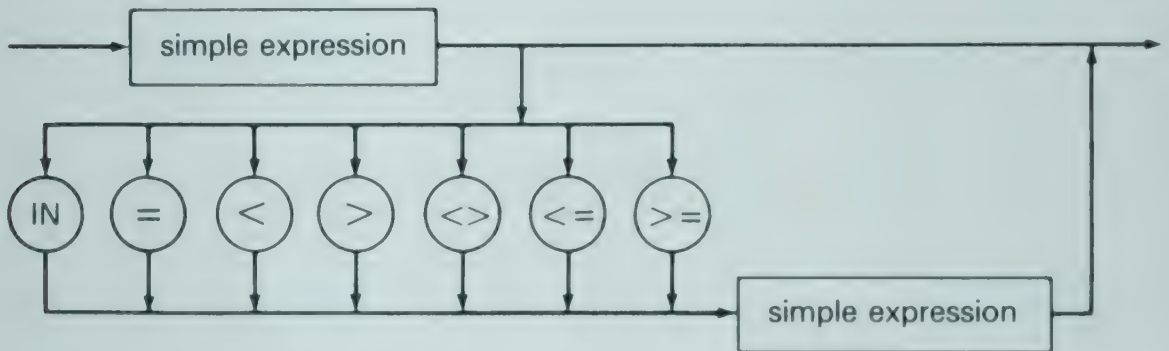
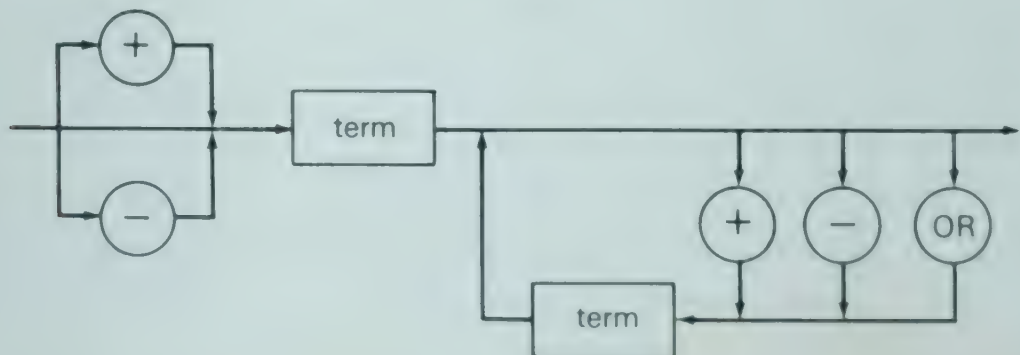
Note: The order of presentation of the diagrams is from higher-level constructs to lower-level constructs. Thus we begin with the diagram for *program*, then proceed to *block*, and so forth, ending with the diagram for *identifier*.

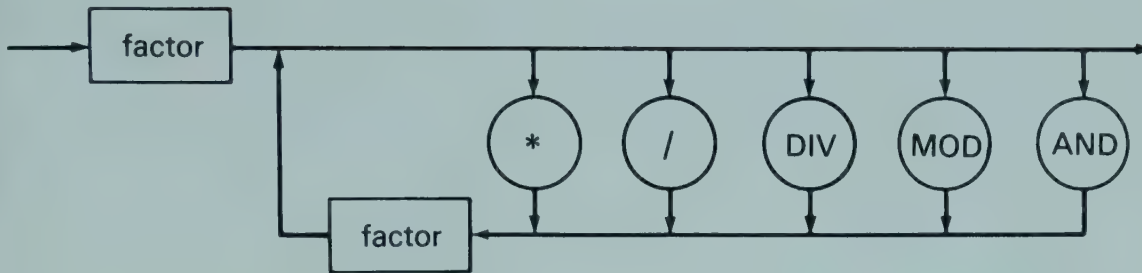
Program



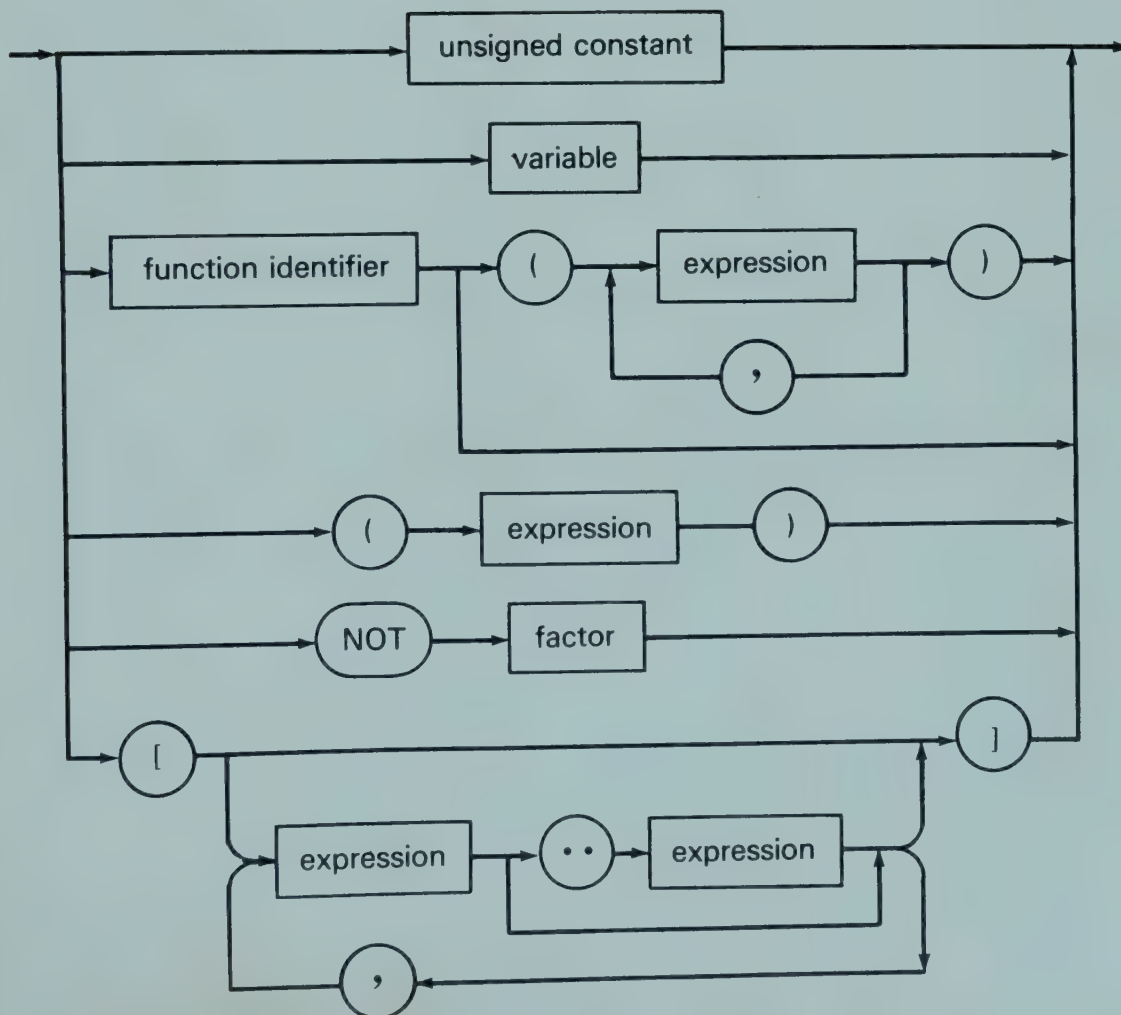


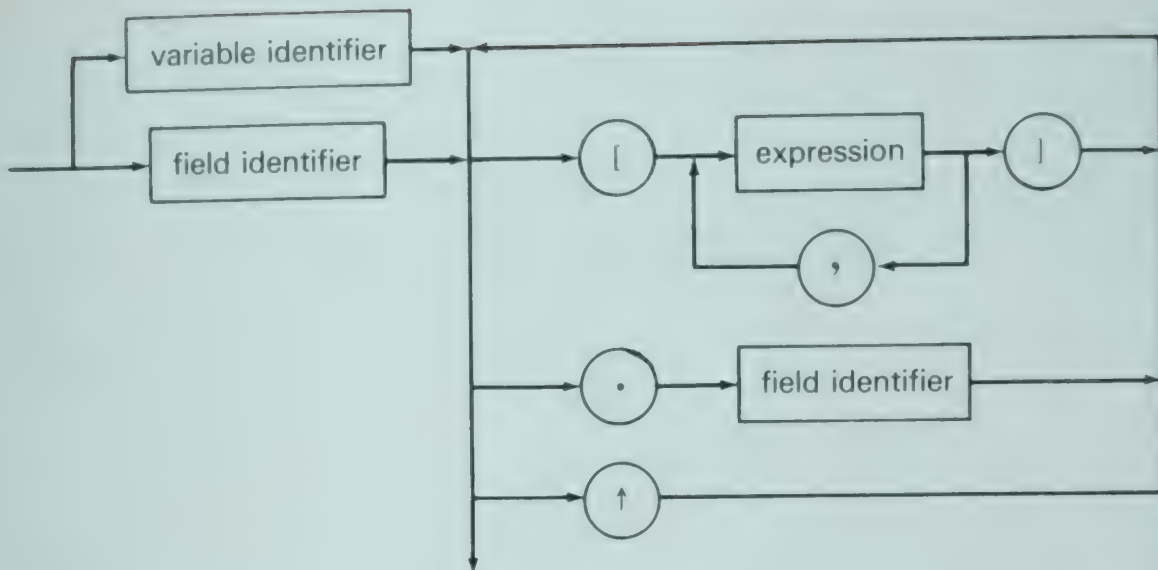


*Expression**Simple Expression*

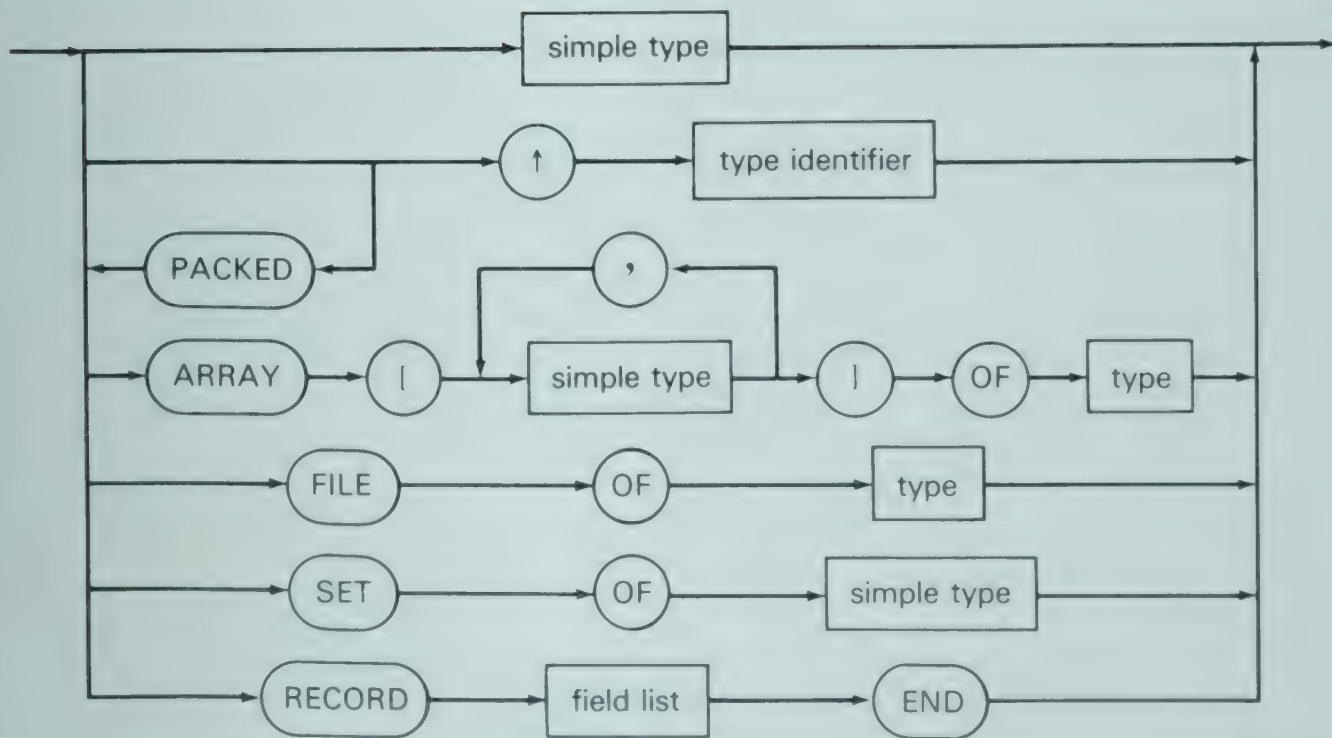


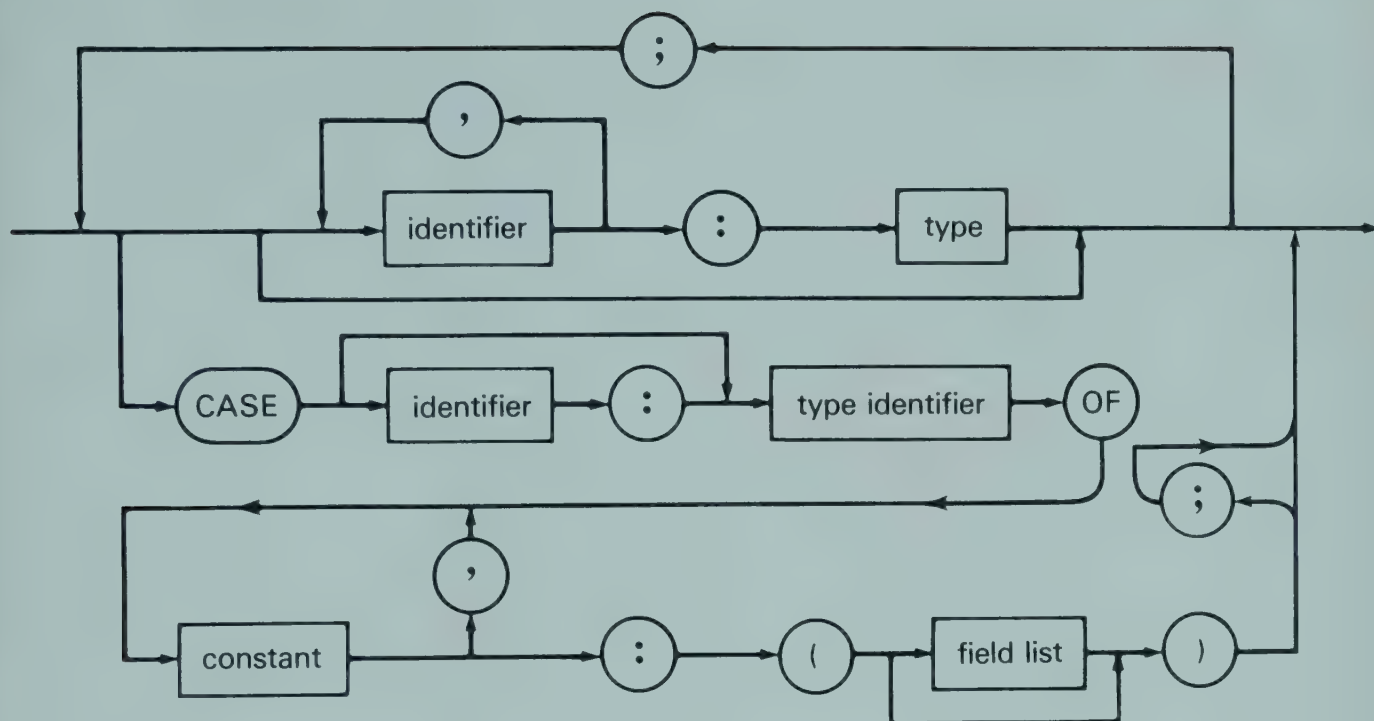
Factor



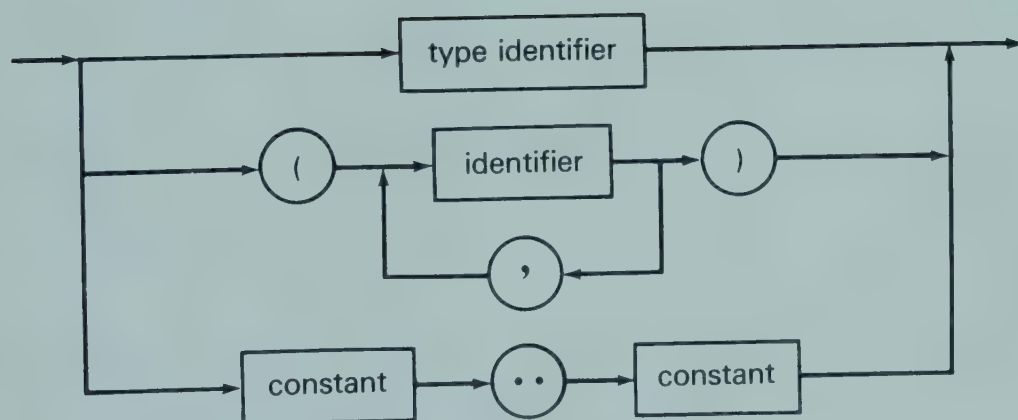


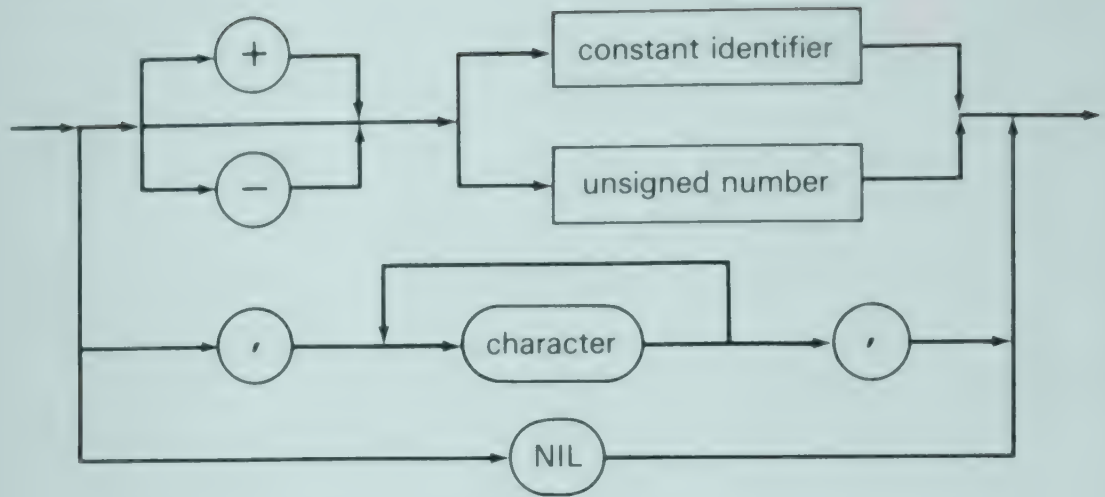
Type



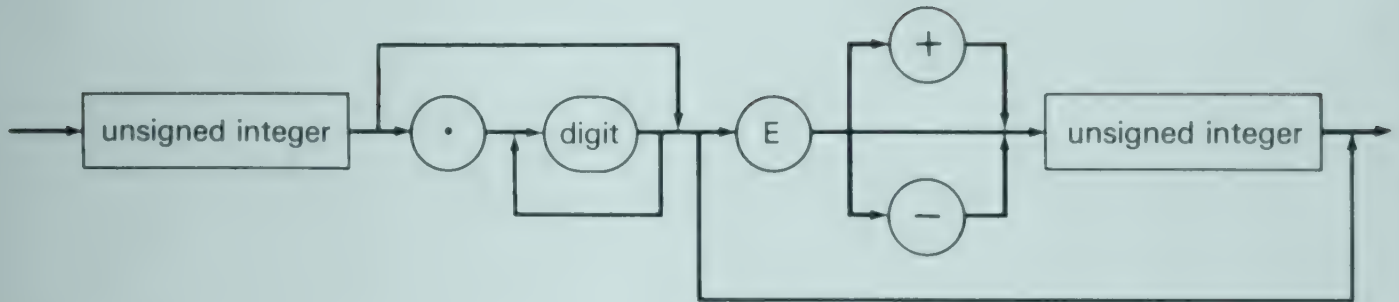


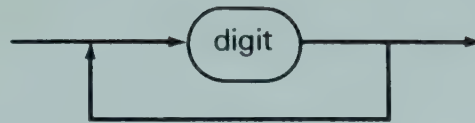
Simple Type



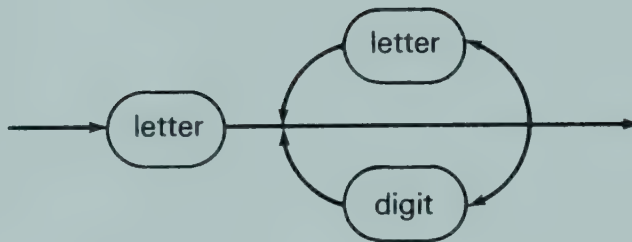


Unsigned Number





Identifier



INDEX

- ABS, 209
- Actual parameters, 187
- ALGOL, 3
- AND, 111, 112
- APL, 3
- ARCTAN, 209
- Arithmetic expressions, 19, 22–25, 122
 - simple, syntax of, 27
 - using INTEGER values, 58–61
- Arithmetic functions, 209
- Arrays, 82–105
 - base address of, 96
 - declaration of, 86–88, 263–265
 - syntax of, 95
 - higher-dimensional, 88–89
 - nested for loops, 89–94
 - packed, 168–170, 208–209
 - two-dimensional, 88
- Assignment statement, 18–22
 - syntax of, 28
- Base address of array, 96
- BASIC, 3
- BEGIN, 7
- Binary operators, 23
- Bisection method, 203
- Blocks, program, 7, 195–200
 - syntax of, 13, 27, 206–207
- BOOLEAN, 110
- BOOLEAN constants, 110, 111
- Boolean operators, 111–112
- BOOLEAN variables, 110–112
- Bound on subscript values:
 - lower, 86
 - upper, 87
- Bubblesort, 149–152
- Buffer variable, 279
- CASE clause, 271
- CASE statements, 141–144, 167

- CASE statements (*Cont.*):
 - syntax of, 145
 - using enumerated types in 245–247
- CHAR constant, 163
- CHAR variables, 163–168
- Character:
 - end-of-file, 285
 - end-of-line, 285
- Character strings, 170–177
- Character variables, 163–168
- CHR, 211
- COBOL, 3
- Collating sequence, 164
- Comments, 30–31
- Compiler, 4
- Compound statements, 70–71
 - syntax of, 78
- CONST, 26
- Constants, 25–26
 - BOOLEAN, 110, 111
 - CHAR, 163
 - string, 170
 - syntax of, 177
- Control variable, 66
- COS, 209
- Debugging, program, 103–105, 214–215
- Default fieldwidth:
 - for INTEGER number, 61–62
 - for REAL number, 38
- Defaults, 9
- Devices:
 - input, 33
 - output, 8
- Difference of sets, 257
- DISPOSE, 306, 307
- DIV, 60
- Documentation, 48
- Efficiency, 95–98
- Element of a set, 256
- ELSE, 134, 135, 137, 139
- END, 7
- End-of-file character, 285
- End-of-file marker, 279
- End-of-line character, 285
- Enumerated types, 241–243
 - in CASE statements, 245–247
- EOF, 279
- EOLN, 287
- Executive, 5
- EXP, 209
- Exponential representation, 55
- Expression(s):
 - arithmetic (*see* Arithmetic expressions)
 - simple, 121–122
 - syntax of, 121
- External files, 284
- Factor, syntax of, 28, 207
- FALSE, 110–112, 114–115
- Field list, 269, 271
 - syntax of, 289
- Fields:
 - output, 37
 - of records, 269, 271–277
- Fieldwidth (*see* Default fieldwidth)
- Files, 269, 277–288
 - external, 284
 - internal, 284
 - sequential, 278
 - syntax for defining, 288
 - textfiles, 285–288
- Final value of loop index, 66, 73
- Floating point, 37
- Flowcharts, 75–77, 138–139
- FOR statement, 66–79
 - nested, 89–94
 - syntax of, 77
- Formal parameters, 187
- Format, output, 35, 61
- FORTRAN, 3
- Forward reference, 230
- Fractions, 54
- Functions, 183, 200–205
 - arithmetic, 209
 - ordinal, 210–212
 - as parameters, 202–205
 - syntax for defining, 205
- GET, 281
- Global names, 191
- Global variables, 191–195
- Hardware, 5
- Higher dimensional array, 88–89
- Identifiers:
 - scope of, 198
 - syntax of, 12
- IF statement, 134–141
 - nested, 139–141, 145–146

- IF statement (*Cont.*):
 - syntax of, 144
- Implementation-dependent, 4
- Index, 66
- Index variable, 66
- Infinite loops, 72, 125–126
- Initial value of loop index, 66, 73
- Input with READ and READLN, 42–46
- Input devices, 33
- INTEGER, 53–63
- Integer representation, 55
- INTEGER values:
 - arithmetic expressions using, 58–61
 - writing, 61–62
- Interactive programs, 46–48
- Internal file, 284
- Intersection of sets, 257
- Language(s):
 - machine, 2
 - programming, 3–4
- Linked lists, 304–308
- LN, 209
- Local name, 191, 194–195
- Local variable, 191–192
- Loops:
 - FOR (*see* FOR statement)
 - infinite, 72, 125–126
 - REPEAT, 118–120
 - WHILE, 108–109
- Lower bound of array subscript, 86
- Machine language, 2
- Marker, end-of-file, 279
- Member of a set, 256–257
- Memory, 17, 96
- MOD, 60–61
- Modular program design, 183
- Monitor, 5
- Names:
 - global, 191
 - local, 191, 194–195
 - program, 6–7
 - scope of, 198
- Nested FOR loops, 89–94
- Nested IF statements, 139–141, 145–146
- NEW, 302, 307
- NIL, 306
- NOT, 111, 112
- Notation, scientific, 37
- Null statement, 137
- Number, ordinal, 210–211, 244
- Operators:
 - binary, 23
 - Boolean, 111–112
 - relational, 112–115
- OR, 111, 112
- ORD, 210, 211, 244
- Ordinal functions, 210–212
- Ordinal number, 210–211, 244
- Ordinal types, 244
- Output:
 - with WRITE, 40–42
 - with WRITELN, 34–40
- Output devices, 8
- Output format, 35, 61
- PACK, 208–209
- Packed arrays, 168–170, 208–209
- Parameter list, 186
 - syntax of, 206
- Parameters:
 - actual, 187
 - formal, 187
 - subprogram, 265–266
 - value, 188–190
 - variable, 188–190
- PL/1, 3
- Pointers, 301–304
 - syntax for defining, 308
- Precedence, rules of, 24
- PRED, 211, 244
- Predecessor function, 211, 244
- Procedure statements, 207
- Procedures, 183–200, 202–205, 207–209
 - block-structured programs, 195–200
 - local and global names, 191–195
 - syntax for defining, 205
 - value parameters, 185–188
 - variable parameters, 188–189
 - (*See also* Functions)
- Program, 3
 - query, 294
 - report-writing, 294
 - syntax of, 13
- Program name, 6–7
- Program specifications, 126–127
- Program testing, 123–124, 147–148
- Programming languages, 3–4
- Proper subsets, 257
- PUT, 280

Query programs, 294
Quicksort, 233–234

READ, 42–46, 166
READLN, 42–46
REAL, 22, 38, 57–61
Records, 268–277
 declaring, 269–273
 defining types, 269–273
 referencing, 273–277
 syntax for defining, 288
Recursive call, 220
Recursive subprograms, 219–238
Reference, forward, 230
Relational condition, 113–114
Relational operators, 112–115
REPEAT statement, 118–120
 syntax of, 121
Report-writing program, 294
Representation:
 exponential, 55
 integer, 55
Reserved words, 7
RESET, 281–282
REWRITE, 282
ROUND, 60–61
Rules of precedence, 24

Scientific notation, 37
Scope:
 of identifiers, 198
 of names, 198
 of variables, 198
Sequence, collating, 164
Sequential files, 278
SET, syntax for defining, 266
Set operations, 259–263
Set theory, elementary, 256–257
Sets:
 mathematical, 255
 in Pascal, 257–259
Side effects, avoiding, 212–214
Simple arithmetic expression, syntax of,
 27
Simple expression, 121–122
SIN, 209
Software, 5
Sorting, 304
 programming examples of, 148–152,
 232–238
SQR, 209
SQRT, 209

Statements:
 assignment, 18–22
 syntax of, 28
 CASE (see CASE statements)
 compound, 70–71
 syntax of, 78
 FOR (see FOR statement)
 IF (see IF statement)
 null, 137
 procedure, 185
 (See also Procedures)
 REPEAT, 118–120
 syntax of, 121
 WHILE, 108–109
 syntax of, 121
 WITH, 274
 syntax of, 289
String constants, 170
String variables, 170–175
Strings, character, 170–177
Subprograms, 183
 recursive, 219–238
Subrange types, 247–248
Subscript, 83
Subscript values (see Bound on subscript
 values)
Subscripted variables, 83–86
Subsets, 257
 proper, 257
SUCC, 211, 244
Successor function, 211, 244
Syntax, 11–14
 (See also *specific statement or
 expression*)
Syntax diagrams, 11–14
 (See also *specific statement or
 expression*)

Tag field, 271–272
Term, syntax of, 28, 62, 122
Termination condition for recursive
 program, 221
Text editing, 309–313
Textfiles, 285–288
THEN, 134–139
TRUE, 110–115
TRUNC, 60–61
Two-dimensional array, 88
Type conversions, 56–58
Type definition, syntax of, 249
Type errors, 56–58
Types:
 simple, 241–252

Types, simple (*Cont.*):

- enumerated, 241–243
- ordinal, 244
- subrange, 247–248
- syntax of, 249
- structured, 241
 - sets and arrays as, 255–266

Unary operator, 23

Union of sets, 257

Unknown amounts, reading, 115–118

UNPACK, 208

Upper bound of array subscript, 87

Validity checking, 49–50, 137

Value of loop index:

- final, 66
- initial, 66

Value parameters, 188–190

VAR, 22

Variable parameters, 188–190

Variables, 19

- BOOLEAN, 110–112

Variables (*Cont.*):

- buffer, 279
- CHAR, 163–168
- control, 66
- global, 191–195
- index, 66
- local, 191–192
- naming, 29–30
- scope of, 198
- string, 170–175
- subscripted, 83–86
- syntax of, 95, 289

Variant parts of records, 271–272

WHILE statement, 108–109

- syntax of, 121

Wirth, Niklaus, 4

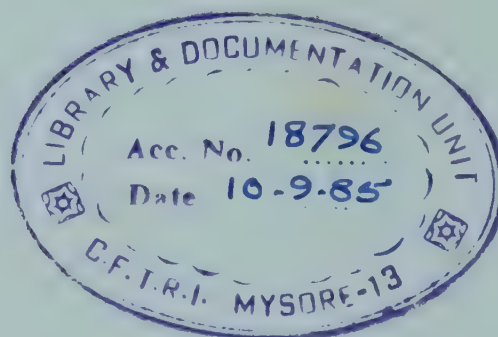
WITH statement, 274

- syntax of, 289

Words, reserved, 7

WRITE, 40–42, 166

WRITELN, 8–11, 13–15, 34–40



Central Food Technological Research
Institute.

12-2-89
11/30/89

C.F.T.R.I. LIBRARY, MYSORE - 13

12597

Acc.No. 18796

Call No. 98 N84

CHECKED
2008
2

Please return this publication on or before the last DUE
DATE stamped below to avoid incurring overdue charges.
To be issued from 6.11.1985

VERIFIED
2013
8

Due date	Return Date	Due Date	Return Date
~~25-11-86~~		~~11.11.91~~	
~~21-7-87~~	~~27-7-87~~	7.8.92	3-8.
		11/10/92	21.10.
~~13-8-87~~	~~15-8-87~~	5.11.92	5.11.92
~~14-9-87~~	~~15-6-87~~	242	
~~30-9-87~~		5.4.73	6/4
		15.4.95	28.4.95
18.6.88	14.6.88	13.5.95	13.6.91
		28.6.95	9/8/95
~~21-3-90~~		6.10.95	27-10-95
24.4.91	21.6.91	22/14/95	11/12/93
~~9-5-91~~	21/5/92	21/5/92	
~~5-11-91~~		5/6/96	4/6/90
~~29-11-91~~		19/6/96	14/6

CFTRI-MYSORE



18796
Pascal.

SV2019

\$24 - 95 #24-95

80341

301184

Acc. No. 18796

N86

FORD (GG)

al.

ISBN 0-07-038138-0